DELHI TECHNOLOGICAL UNIVERSITY

THEORY OF COMPUTATION PROJECT REPORT

Turing Machine:

The Busy Beaver problem

SUBMITTED BY:

KUMAR APURVA (DTU/2K18/MC/058)

MADHURESH MAYANK (DTU/2K18/MC/062)

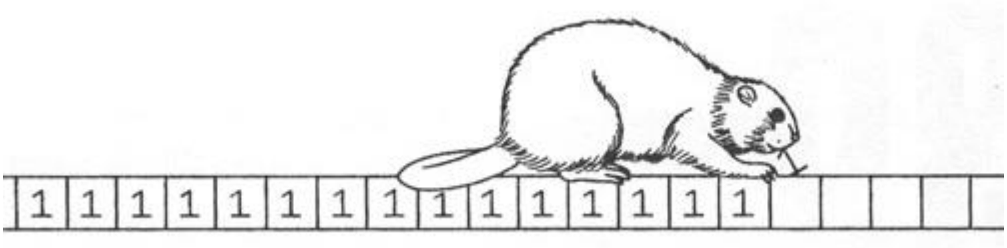
INDEX

INTRODUCTION

The **Busy Beaver** problem is a fun theoretical computer science problem. Intuitively, the problem is to find the smallest program that outputs as many data as possible and eventually halts. Originally the only Turing Machines considered were those using only 2 symbols (0 and 1).

The two functions defined below, called **Busy Beaver functions (Σ(n)) and Maximum shifts function (S(n))**, were introduced in 1962 by Tibor Rado as simple examples of non computable functions:

1. **Σ(n)**: the largest number of 1's printable by an n-state machine before halting, and
2. **S(n)**: the largest number of steps taken by an n-state machine before halting.



Both of these functions are non computable, because they grow faster than any computable function. Even with only a few states, a Busy Beaver can do very much.

The current 5-state Busy Beaver champion produces 4,098 ones, using 47,176,870 steps.

Generalization

For any model of computations there exist simple analogs for Busy Beaver.

The generalization to Turing Machines with n states and m symbols defines the following **generalized busy beaver functions**:

1. **Σ(n,m)**: the largest number of non-zeros printable by an n-state, m-symbol machine before halting, and

2. **S(n,m)**: the largest number of steps taken by an n-state, m-symbol machine before halting.

The longest running 3-state 3-symbol machine found so far (found by Allen Brady) runs 92,649,163 steps before halting.

Properties of Function S and Σ

1. These functions grow faster than any computable function. Formally, for any computable function f, there is an integer N such that, for any integer n > N,

   $$S(n) > \Sigma(n) > f(n).$$

   This was proved by Tibor Rado who defined these functions in order to get non computable functions.

2. It is easy to prove that the two variables functions S(n,m) and Σ(n,m) are increasing with the number n of states if the number m of symbols is constant. Formally, for any integer m, if n > k, then

   $$S(n,m) > S(k,m) \quad \text{and} \quad \Sigma(n,m) > \Sigma(k,m).$$

Relation between S and Σ

Many mathematicians tried to figure out the relation between S and Σ.

- Rado (1962) proved that $S(n) < (n+1)\ \Sigma(5n)*2^{\Sigma(5n)}$.
- Julstrom (1993) proved that S(n) < Σ(28n).
- Julstrom (1992) proved that S(n) < Σ(20n).
- Wang and Xu (1995) proved that S(n) < Σ(10n).
- Buro (1990) proved that S(n) < Σ(9n).
- Yang, Ding and Xu (1997) proved that S(n) < Σ(8n), and that there is a constant c such that S(n) < Σ(3n+c).
- Ben-Amram, Julstrom and Zwick (1996) proved that S(n) < Σ(3n+6),and
  S(n) < (2n-1) Σ(3n+3).
- Ben-Amram and Petersen (2002) proved that there is a constant c such that
  $S(n) < \Sigma(n + 8n/\log_2 n + c)$.

BACKGROUND KNOWLEDGE

TURING MACHINES

For understanding Busy Beaver problem, we first need to know about Turing Machines. So, what is a Turing Machine?

Turing Machine was invented by Alan Turing in 1936 and it is used to accept Recursive Enumerable Languages (generated by Type-0 Grammar).

A Turing Machine consists of a tape of infinite length on which read and writes operation can be performed. The tape consists of infinite cells on which each cell either contains input symbol or a special symbol called blank. It also consists of a head pointer which points to cell currently being read and it can move in both directions.

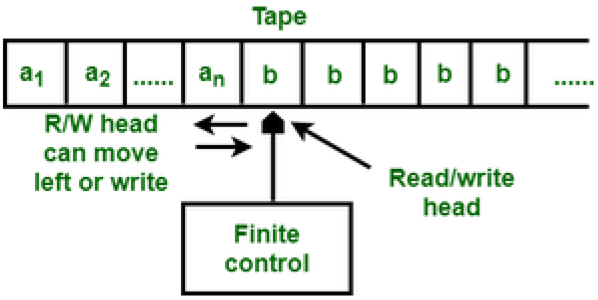


Illustration of a Turing Machine

A Turing Machine is expressed as a 7-tuple (Q, T, B, ∑, δ, q0, F) where:

- **Q** is a finite set of states
- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- **∑** is the input alphabet (symbols which are part of input alphabet)
- **δ** is a transition function which maps Q × T → Q × T × {L,R}. Depending on its present state and present tape alphabet (pointed by head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.
- **q0** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

COMPUTABILITY AND UNCOMPUTABILITY

A real number *a* is computable if it can be approximated by some computable function from the natural numbers **N** to the real numbers **Z** which, given any positive integer *n*, the function produces an integer *f(n)* such that:

$$\frac{f(n)-1}{n} \leq a \leq \frac{f(n)+1}{n}$$

Turing's definition of computability (1936)
A function is intuitively computable (effectively computable) if and only if it is computable by a Turing machine; that is, an automatic machine.

Archimedes' constant (pi), along with other well-known numbers such as Pythagoras' constant (√2) and the golden ratio (ϕ) are all examples of a type of real number which we say is computable, despite also being irrational (real numbers which cannot be constructed from fractions of integers). Such computable numbers may be defined as real numbers that can be computed to within any desired precision by a finite, terminating algorithm. More Formally we can define as *A number for which there is a Turing machine which, given n on its initial tape, terminates with the nth digit of that number* [encoded on its tape].

A function which is not computable is termed as uncomputable. More Formally, A function that cannot be computed by any algorithm --- equivalently, not by any Turing machine.

Non-Computable Problems

Non-computablity is a problem for which there is no algorithm that can be used to solve it. Most famous example of a non-computablity (or undecidability) is the **Halting Problem**. Given a description of a Turing machine and its initial input, determine whether the program, when executed on this input, ever halts (completes).

The alternative is that it runs forever without halting. The halting problem is about seeing if a machine will ever come to a halt when a certain input is given to it or if it will finish running. This input itself can be something that keeps calling itself forever which means that it will cause the program to run forever.

Approach and Implementation

Approach

We are building Busy Beaver with 2 symbols (m=2) and 4 different states (n=1, 2, 3 and 4). We are considering the 2 symbols as 0 and 1 and the starting state as “q_1“ and halting state as "H".

The machine's *transition function* takes two inputs:

1. the current non-Halt state,
2. the symbol in the current tape cell,

and produces three outputs:

1. a symbol to write over the symbol in the current tape cell (it may be the same symbol as the symbol overwritten),
2. a direction to move (left or right; that is, shift to the tape cell one place to the left or right of the current cell), and
3. a state to transition into (which may be the Halt state).

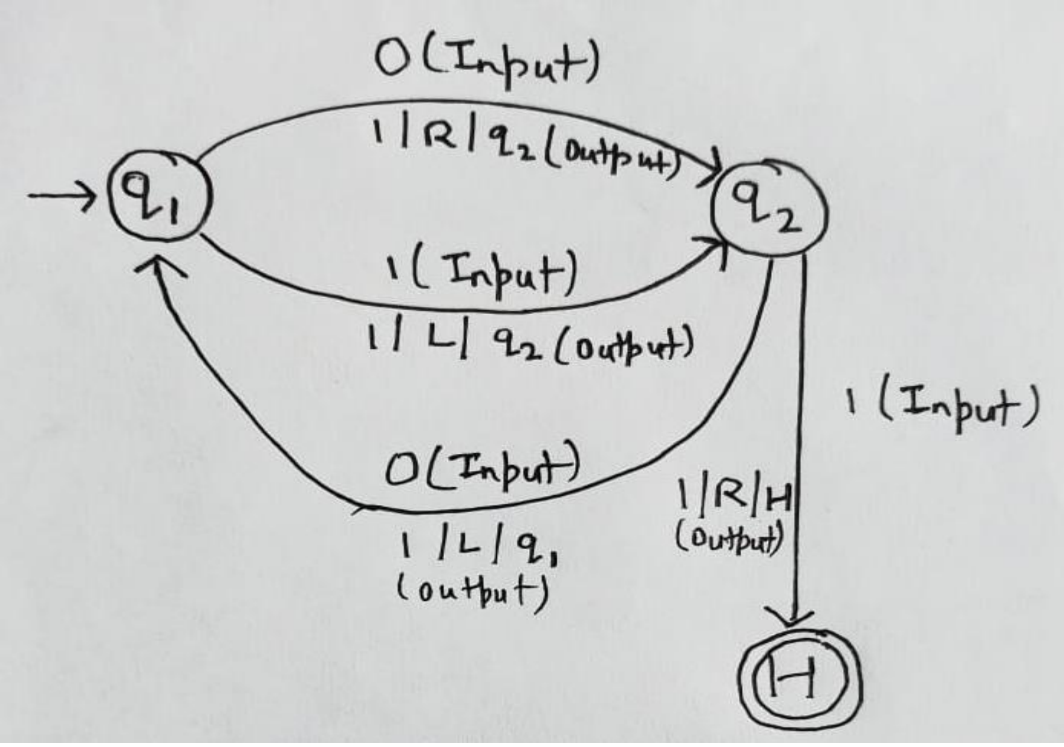
The general formula to compute the number of Turing machines for n state m symbols is

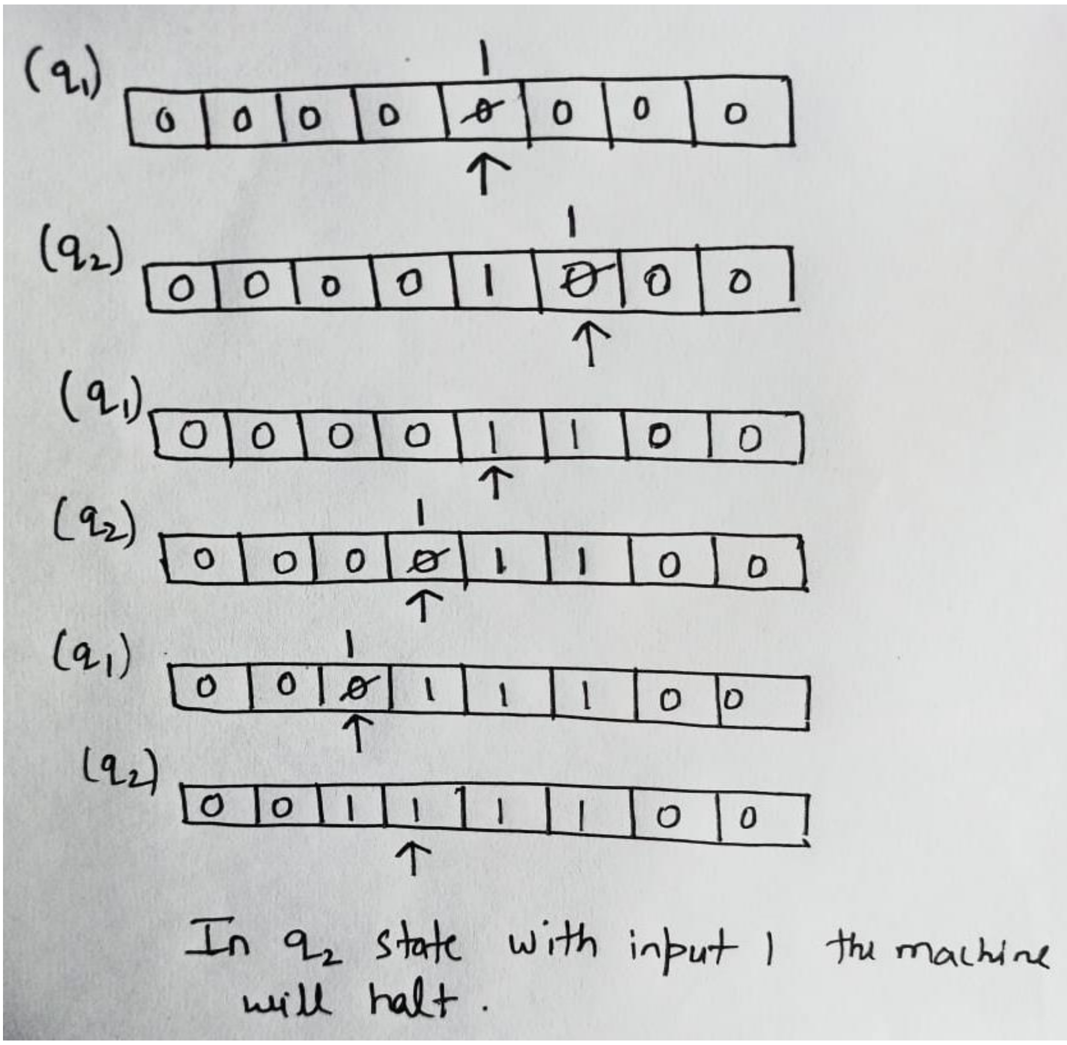
$$(symbols * directions * (states + 1))^{(symbols * states)}$$

For a n state 2 symbol Busy Beaver there are $\mathbf{(4n + 4)^{2n}}$ Turing machines and we need to check all these machines to get our desired result.

The tapes of Turing Machines are initially filled with 0's. We are interested in all those turing machines which will finally halt and we need to find the best machine i.e. the machine which prints the most number of 1’s in the tape before it halts and the number of steps taken by that machine.

Below we have made transition diagram and their corresponding tape cell for each step of two Busy Beaver Turing Machines i.e 2 State 2 Symbol and 3 State 2 Symbol machines. We are considering those machines which gave the best result for the corresponding pair of state and symbol.





(q1)

0	0	0	0	~~0~~ 1	0	0	0
				↑			

(q2)

0	0	0	0	1	~~0~~ 1	0	0
					↑		

(q1)

0	0	0	0	1	1	0	0
				↑			

(q2)

0	0	0	~~0~~ 1	1	1	0	0
			↑				

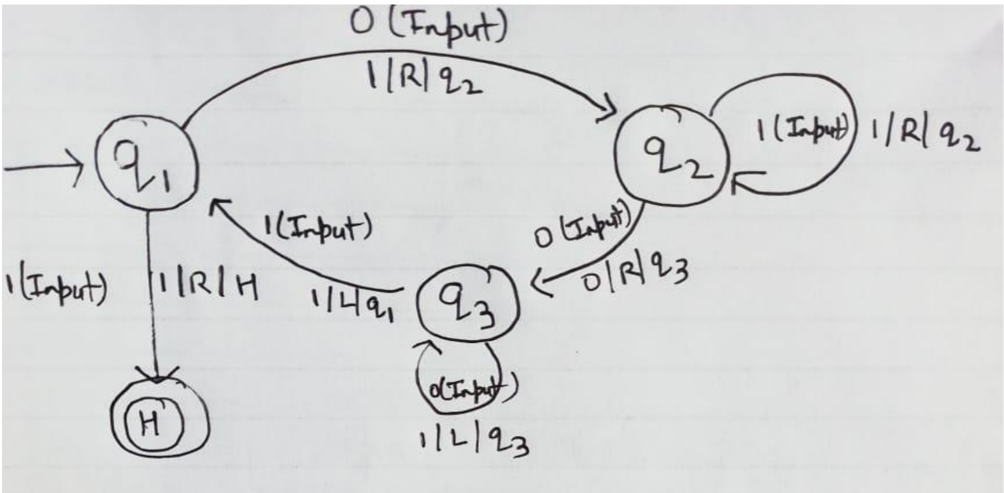
(q1)

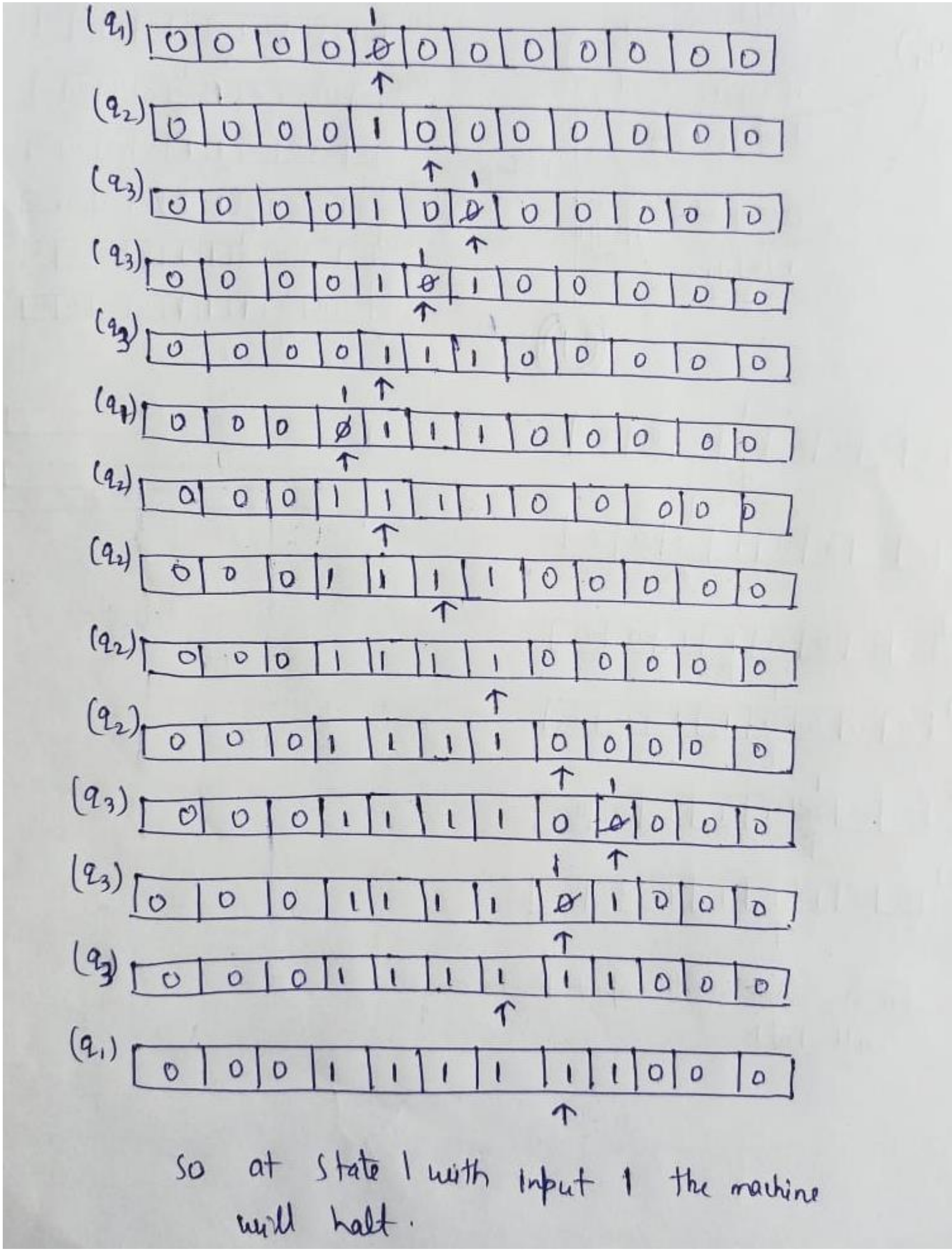
0	0	~~0~~ 1	1	1	1	0	0
		↑					

(q2)

0	0	1	1	1	1	0	0
			↑				

In q_2 state with input 1 the machine will halt.



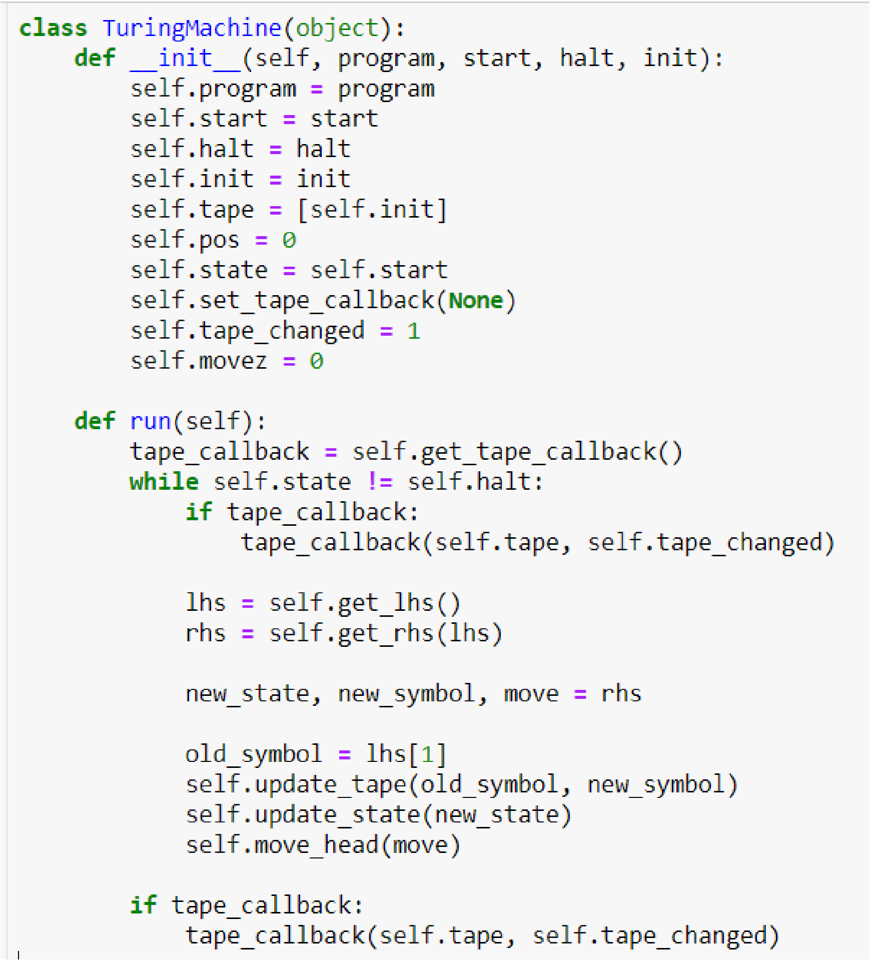


(q_1)	0	0	0	0	~~0~~ 1	0	0	0	0	0	0	0
(q_2)	0	0	0	0	1	0	0	0	0	0	0	0
(q_3)	0	0	0	0	1	0	~~0~~ 1	0	0	0	0	0
(q_3)	0	0	0	0	1	~~0~~ 1	1	0	0	0	0	0
(q_3)	0	0	0	0	1	1	1	0	0	0	0	0
(q_1)	0	0	0	~~0~~ 1	1	1	1	0	0	0	0	0
(q_2)	0	0	0	1	1	1	1	0	0	0	0	0
(q_2)	0	0	0	1	1	1	1	0	0	0	0	0
(q_2)	0	0	0	1	1	1	1	0	0	0	0	0
(q_2)	0	0	0	1	1	1	1	0	0	0	0	0
(q_3)	0	0	0	1	1	1	1	0	~~0~~ 1	0	0	0
(q_3)	0	0	0	1	1	1	1	~~0~~ 1	1	0	0	0
(q_3)	0	0	0	1	1	1	1	1	1	0	0	0
(q_1)	0	0	0	1	1	1	1	1	1	0	0	0

So at state 1 with input 1 the machine will halt.

Implementation

We have implemented the Busy Beaver Function using the above mentioned approach. Our code prints changes done on the tape for each step and prints the final condition of the tape before halting for the busiest beaver machine and from this we can count the total no. of 1's the tape finally have. We are also outputing the total number of steps taken by the busiest beaver machine. Below are some of the snippets from our code.



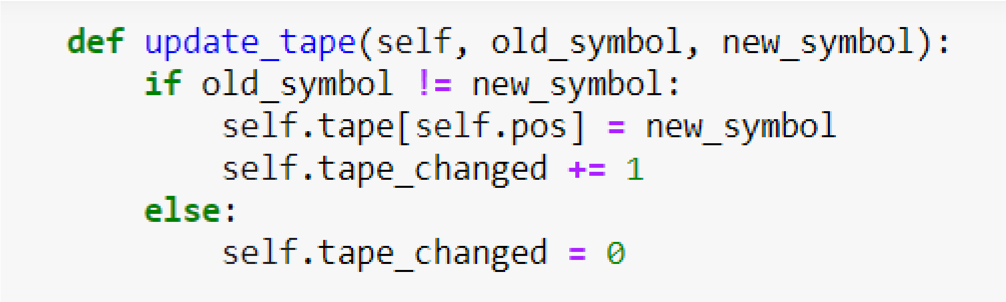
```
class TuringMachine(object):
    def __init__(self, program, start, halt, init):
        self.program = program
        self.start = start
        self.halt = halt
        self.init = init
        self.tape = [self.init]
        self.pos = 0
        self.state = self.start
        self.set_tape_callback(None)
        self.tape_changed = 1
        self.movez = 0

    def run(self):
        tape_callback = self.get_tape_callback()
        while self.state != self.halt:
            if tape_callback:
                tape_callback(self.tape, self.tape_changed)

            lhs = self.get_lhs()
            rhs = self.get_rhs(lhs)

            new_state, new_symbol, move = rhs

            old_symbol = lhs[1]
            self.update_tape(old_symbol, new_symbol)
            self.update_state(new_state)
            self.move_head(move)

        if tape_callback:
            tape_callback(self.tape, self.tape_changed)
```

The **run** function is used for each Turing machine corresponding to n state and it will stop when we reach the halt state.

lhs consist of the current state and the current symbol corresponding to the header and **rhs** will store the new,state, new symbol and the move(i.e. left or right) corresponding to the lhs.

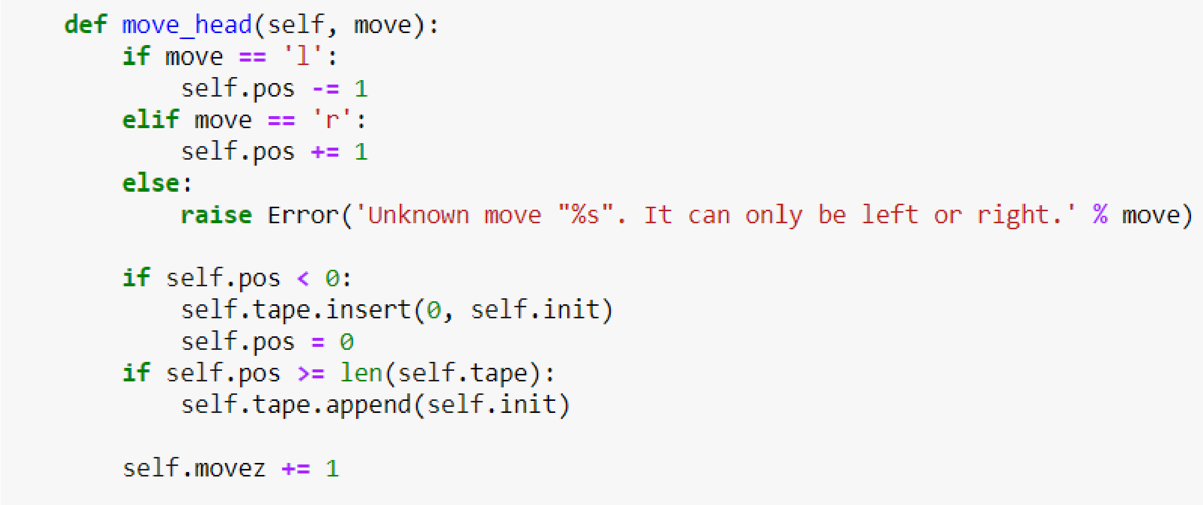
We then update the symbol on the tape corresponding to the head.

After this we move the head to the next positon.



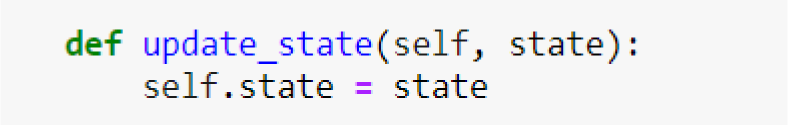
```
def update_tape(self, old_symbol, new_symbol):
    if old_symbol != new_symbol:
        self.tape[self.pos] = new_symbol
        self.tape_changed += 1
    else:
        self.tape_changed = 0
```

This function performs the Updation and it changes the old symbol to the new symbol if old symbol and new symbol are not equal.



```
def move_head(self, move):
    if move == 'l':
        self.pos -= 1
    elif move == 'r':
        self.pos += 1
    else:
        raise Error('Unknown move "%s". It can only be left or right.' % move)

    if self.pos < 0:
        self.tape.insert(0, self.init)
        self.pos = 0
    if self.pos >= len(self.tape):
        self.tape.append(self.init)

    self.movez += 1
```

This function is used to move the head to left or right position.



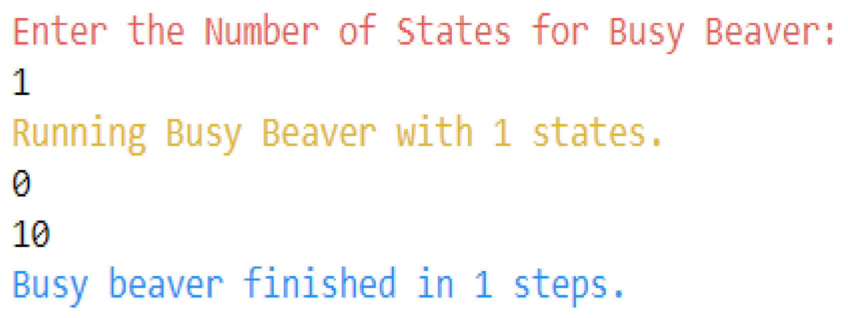
```
def update_state(self, state):
    self.state = state
```

This function updates the state of the machine.

RESULT AND ANALYSIS

This is the result generated by our implemented code which prints the tape condition at each step before it halts and total steps taken by the busiest beaver machine. We ran our code for 4 different state(n) and 2 symbols (0 and 1).

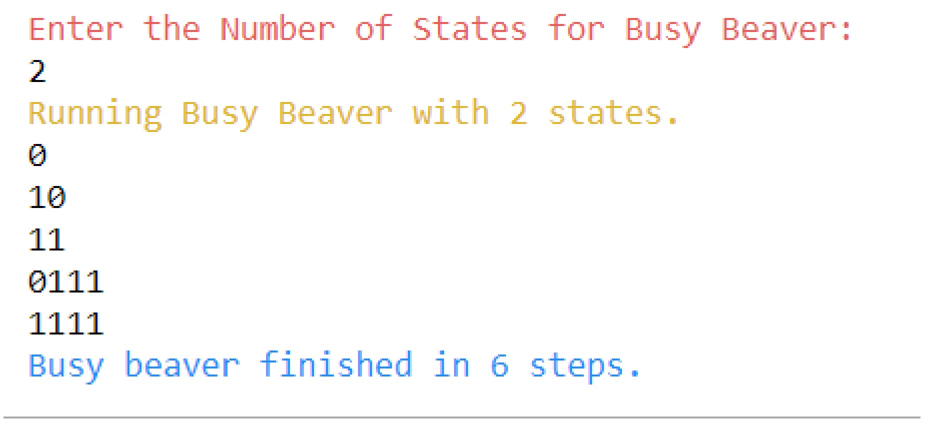
1. For a 1 state 2 symbol Turing Machine



```
Enter the Number of States for Busy Beaver:
1
Running Busy Beaver with 1 states.
0
10
Busy beaver finished in 1 steps.
```

In our output for 1 state 2 symbol busy beaver we can see that the best machine or the busiest beaver finally consist of one 1 on the tape and it required 1 step to achieve it.

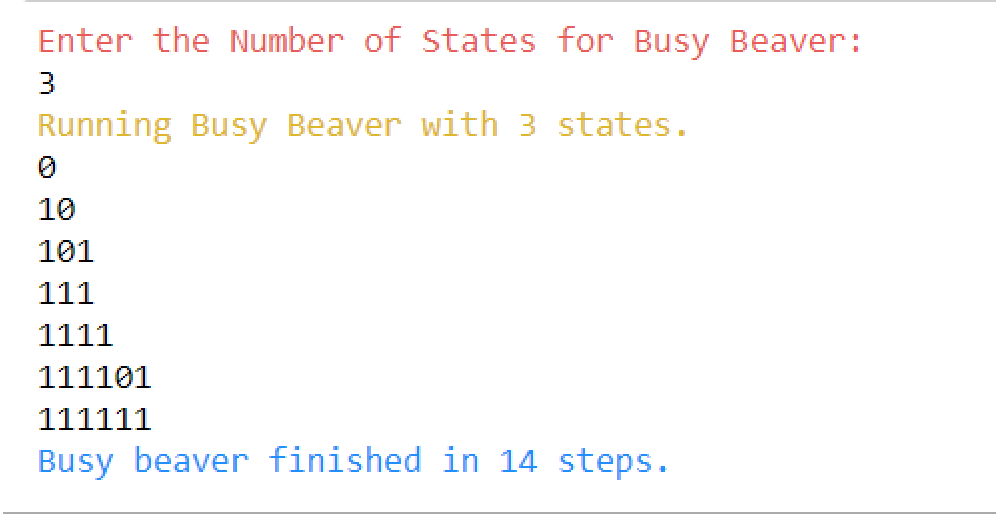
2. For a 2 state 2 symbol Turing Machine



```
Enter the Number of States for Busy Beaver:
2
Running Busy Beaver with 2 states.
0
10
11
0111
1111
Busy beaver finished in 6 steps.
```

In our output for 2 state 2 symbol busy beaver we can see that the best machine or the busiest beaver finally consist of four 1's on the tape and it required 6 step to achieve it.

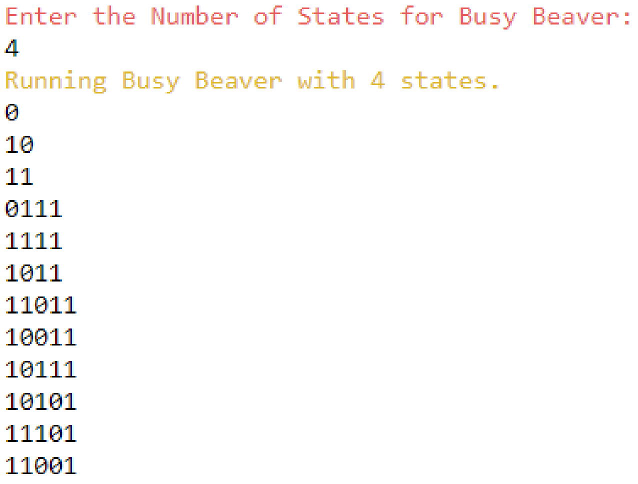
3. For a 3 state 2 symbol Turing Machine



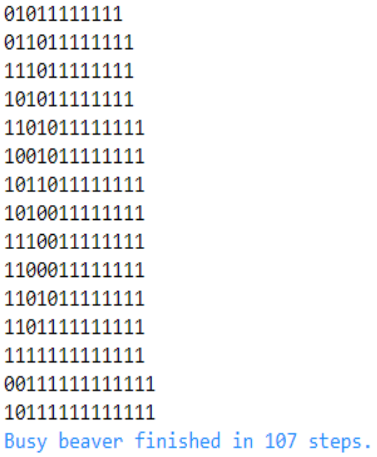
```
Enter the Number of States for Busy Beaver:
3
Running Busy Beaver with 3 states.
0
10
101
111
1111
111101
111111
Busy beaver finished in 14 steps.
```

In our output for 3 state 2 symbol busy beaver we can see that the best machine or the busiest beaver finally consist of six 1’s on the tape and it required 14 step to achieve it.

4. For a 4 state 2 symbol Turing Machine



```
Enter the Number of States for Busy Beaver:
4
Running Busy Beaver with 4 states.
0
10
11
0111
1111
1011
11011
10011
10111
10101
11101
11001
```



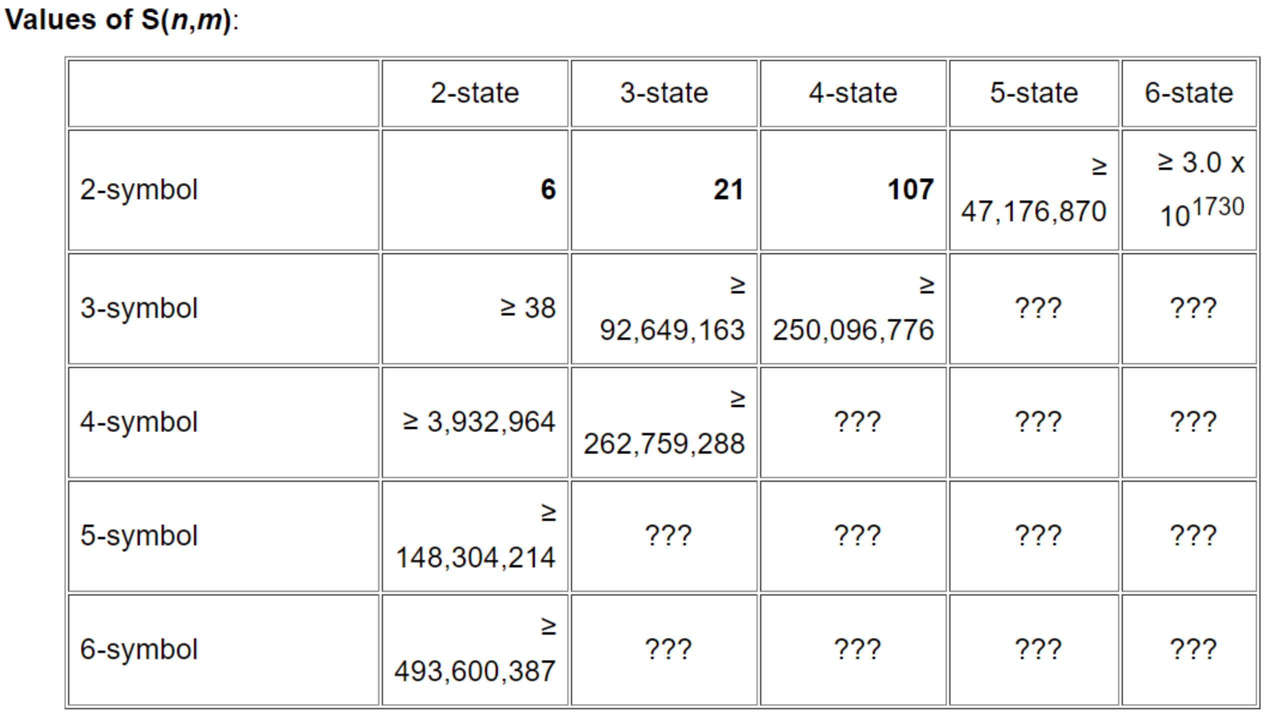
```
01011111111
011011111111
111011111111
101011111111
1101011111111
1001011111111
1011011111111
1010011111111
1110011111111
1100011111111
1101011111111
1101111111111
1111111111111
00111111111111
10111111111111
Busy beaver finished in 107 steps.
```

As the output is large we are just showing some part of the output.

In our output for 4 state 2 symbol busy beaver we can see that the best machine or the busiest beaver finally consist of thirteen 1’s on the tape and it required 107 step to achieve it .

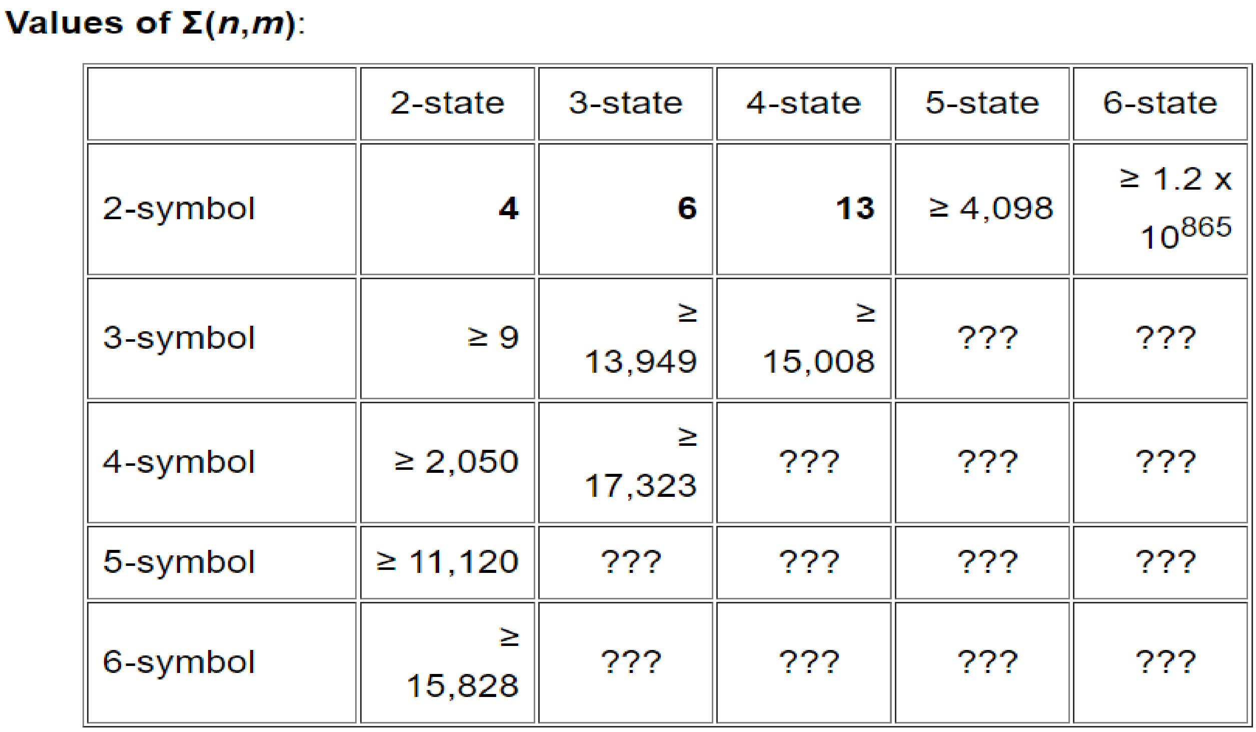
Now, we have shown the output corresponding to four different pairs of n and m. These values were easily computable from our code but for **other pairs the step values turns out to be very large and these are uncomputable till date**. Mathematicians have tried to work on these and they have computed a **lower bound** for some of the pairs of n and m which we can see in the table below.

The following table lists the exact values and some known lower bounds for **S(n,m)** and **Σ(n,m)** for the generalized busy beaver problems. Known exact values are shown in bold face type and known lower bounds are preceded by a greater than or equal to (≥) symbol. The entries listed as "???" are bounded by the maximum of all entries to left and above but have not been explicitly investigated by anyone to date.



Values of S(*n*,*m*):

	2-state	3-state	4-state	5-state	6-state
2-symbol	**6**	**21**	**107**	≥ 47,176,870	≥ 3.0 x 10^{1730}
3-symbol	≥ 38	≥ 92,649,163	≥ 250,096,776	???	???
4-symbol	≥ 3,932,964	≥ 262,759,288	???	???	???
5-symbol	≥ 148,304,214	???	???	???	???
6-symbol	≥ 493,600,387	???	???	???	???



Values of Σ(*n*,*m*):

	2-state	3-state	4-state	5-state	6-state
2-symbol	**4**	**6**	**13**	≥ 4,098	≥ 1.2 x 10^{865}
3-symbol	≥ 9	≥ 13,949	≥ 15,008	???	???
4-symbol	≥ 2,050	≥ 17,323	???	???	???
5-symbol	≥ 11,120	???	???	???	???
6-symbol	≥ 15,828	???	???	???	???

APPLICATIONS

In addition to posing a rather challenging mathematical game, the busy beaver functions offer an entirely new approach to **solving pure mathematical problems**. Many open problems in mathematics could in theory, but not in practice, be solved in a systematic way given the value of S(n) for a sufficiently large n.

Consider any conjecture that could be disproven via a counterexample among a countable number of cases (e.g. Goldbach's conjecture).

Goldbach's conjecture

Write a computer program that sequentially tests this conjecture for increasing values. In the case of Goldbach's conjecture, we would consider every even number ≥ 4 sequentially and test whether or not it is the sum of two prime numbers. Proving the conjecture true or false would be an epochal event in number theory, allowing mathematicians to better understand the distribution of prime numbers.

Suppose this program is simulated on an n-state Turing machine. It works by counting upward through all even integers greater than 4; for each one, it grinds through all the possible ways to get that integer by adding two others, checking whether the pair is prime. When it finds a suitable pair of primes, it moves up to the next even integer and repeats the process. If it finds an even integer that can’t be summed by a pair of prime numbers, it halts. However, if the conjecture is true, then our program will never halt.

Recent developments in finding Threshold of Unknowability

A 748-state binary Turing machine has been constructed that halts iff ZFC is inconsistent. A 744-state Turing machine has been constructed that halts iff the Riemann hypothesis is false. A 43-state Turing machine has been constructed that halts iff Goldbach's conjecture is false, and a 27-state machine for that conjecture has been proposed but not yet verified.

CONCLUSION AND LEARNING

- We have studied about Turing machine and Busy Beaver in depth.

- We understood how to determine the Transititon diagram for Busy Beaver and how to compute S(n) and ∑(n) corresponding to the best machine.

- We also implemented the Python code to determine both S(n) and ∑(n).

- We tried to understand the real life occurences of Busy Beaver and Turing machines.

- We also developed a firm understanding of computable and uncomputable functions.

REFERENCES

- Busy Beaver - https://en.wikipedia.org/wiki/Busy_beaver
- Busy Beaver Turing Machines – Computerphile – https://www.youtube.com/watch?v=CE8UhcyJS0I
- Busy Beaver Competitions – https://webusers.imj-prg.fr/~pascal.michel/bbc.html
- Computable and Uncomputable Functions – https://en.wikipedia.org/wiki/Computable_function
- Link for the online simulator we have used – https://turingmachine.io/