Practice 1

Abraham Álvarez Cruz

Department of Computer Science
School of Engineering
University of Cádiz

abraham.alvarezcruz@alum.uca.es

28th October 2023

Summary

In this report i will present the results obtained in practice 1 in which we had to work with Turing Machines (TM from now on) to do multiples finite automatons to resolve a variety of problems.
The first two automatons will try two resolve two typical problems when someone learn to create automatons.
The last one will try to reproduce RAM devices using a sequential approach.

Contents

Tables

Figures

1 Introduction

In the first part we will be working with Turing Machines (TM from now on). Turing Machines are theoretical computational machines which can compute functions that exists in the total functions set. There are a plenty examples of functions that belongs to this set. Many of them are: addition, multiplication, division, ... Although almost every function we can think about exists in this set, there are several functions that are not members of that set. Every function that not belong to the total functions set, belongs to the partial functions set. This set contains almost every functions that can exists in this universe.
At first, TM can compute all functions that belongs to total functions set but not the ones that are strictly partial functions. An example of partial function could be subtraction. Although we cannot compute partial functions, that is, return a result for every possible input, we have many tricks to overpass these limits. One possible trick we can do is an universal program that takes: an encoded program, all his possible inputs and a maximum number of steps and, if the function takes less than the specified number of steps to calculate the result, it will return it, otherwise, it will return an special value that indicates the machine couldn't calculate the result.
The problems we will resolve in the next pages will consist in a set of total functions that will return a result for every possible input, that is, our TM's will end all the time in a finite number of steps.
At the end, we will end up with three Turing Machines that compute every problem proposed by the practice guide with his specific analysis telling us *how efficient is our Turing Machine regardless his input*, that is, his *BigO* notation and the results we obtained for a variety of inputs reflected in some charts.
In the second part we will be working with RAM model. RAM model is a bit different from TM in the sense that it can access to every memory position just in one movement unlike TM where we have to access sequentially to the memory to read a specific value from it. We will develop a RAM version of the first two exercises that we'll do.

2 Methods

Our starting point will be a set of TM's previously presented by the teachers. This set of TM's belongs to a Princeton program that we download from [1]. This program is the one we will use to create all our TM's.
The way the programs works it's:

- First, we need to create a specific file in which we create our actual TM, that is: create our possible states, create our transition function specifying *which state we will move to when we read a specific symbol*, the accepting states ....
- Then we will open the program and we'll select our recently created TM.
- The program will show and interface with our TM with all: his vertices, edges...
- Finally, to start using the program we could add an input at the bottom right corner and click on the "play button" at the bottom left corner.

We had a bunch of previously created TM's in the same folder we execute the presented program. We are lucky because many of the examples we had in that folder resolve some of our actual problems. Knowing that, we'll start using them and we'll formalize the actual TM.

2.1 Exercise 1

In the first exercise, we had to do a TM to recognize whether a binary strings contains an equal number of 0's and 1's. I found the program we use had the TM we need to do so i copied it and i will just formalize and explain what it does.

2.1.1 Alphabet

The TM has the following symbols: *0*, *1*, x and $\#$. Symbol x serves to indicate whether the corresponding symbol was processed or not.

2.1.2 Initial state and Terminal states

Our initial point is state *q0* and our accepting state is *q4*.

2.1.3 Transition Table

The next thing that characterizes a TM is his transition table. Our TM has the following specific transition table:

2.1.4 How it works

What the TM actually does is:

- It look's for one 1 or one 0

Actual State	Actual Symbol	New State	New Symbol	Displacement
q0	0|1|x	q0	0|1|x	L
q0	#	q1	#	R
q1	0	q2	x	R
q1	1	q3	x	R
q1	#	q4	#	R
q2	0	q2	0	R
q2	1	q0	x	R
q2	#	q5	#	R
q3	0	q0	x	R
q3	1	q3	1	R
q3	#	q5	#	R

Table 1: Transition table for the first TM.

- If it doesn't find any of these two, it ends returning false (doesn't accept the string.
- If it find any of the mentioned symbols, it goes back to the very start of the string and now, it looks for his counterpart, in other words, if it found an 1 it has to find a 0 and viceversa.

In Figure 1 we can see how our final TM looks in our program.

2.2 Exercise 2

In this second exercise we had to do a TM to tell us if a specific input string contains a number that is multiple of three. As in exercise 1 we will reuse the TM that the program has before hand and i will formalize it and explain what it does.

2.2.1 Alphabet

The TM has the following symbols: *0*, *1* and *#*. As opposed to the exercise one, now we don't have the x symbol because we don't need to use it.

2.2.2 Initial state and Terminal states

Our initial point is state *q0* and our accepting state is *q3*.

2.2.3 Transition Table

The next thing that characterizes a TM is his transition table. Our TM has the following specific transition table:

Actual State	Actual Symbol	New State	New Symbol	Displacement
q0	0	q0	0	R
q0	1	q1	1	R
q0	#	q3	#	R
q1	0	q2	0	R
q1	1	q0	1	R
q1	#	q4	#	R
q2	0	q1	#	R
q2	1	q1	1	R
q2	#	q1	#	R

Table 2: Transition table for the first TM.

2.2.4 How it works

This TM iterates over the entire string (as we did previously with our fist TM) and, depending on our actual state and the symbol we read, we keep track of the remainder we get when we divide our actual number by three. It could be a bit scary at first but it's not that difficult.
To get a better intuition about how it works, i will explain an example. I will use the string "101". If we divide any number by three we can get: 0, 1 or 2 as remainder. In this example our first symbols is one that is, if we now have a 0, the rest will be 2 so it's not multiple of three. If we have a 1 instead, that number is three and three it's multiple of three so the TM will accept the string. We have a 0 so we continue reading the next symbol. Right now, we know that our last two symbols were 1 and 0 so it doesn't matter what symbol we read at this point that it won't be multiple of three because if we read a 0, what we have is number four and if we read a 1, what we have is number 5 so, anyways the number won't be multiple of three.

In Figure 3 we can see how our final TM looks in our program.

2.3 Exercise 3

This exercise tries to recreate what we did in the first exercise with a TM but using the RAM model instead. At high level, what i'm doing is *counting the number of 0's and 1's that i find when iterating the input string.* Every time i see a 0 symbol, i subtract one from our counter and adding one when i see a 1 symbol.
At the end, to return 0 or 1 (false or true) what i do is check whether our counter is equal to zero (equal number of 0's and 1's) or is another quantity.
The code i used to did this exercise will be attached to this document.

2.4 Exercise 4

In the last exercise we have to do the same thing we did in exercise two but using the RAM model. After a long study about how to accomplish our goal effectively trying to take the advantages from RAM model, what i did was mimic the behaviour of the TM i already have using the RAM model. What i actually do is use one register to track our actual state (as we do with our previous TM) and increment and decrement that register depending on our actual state and the symbol we read.
After all the engineering process, what we have is a RAM program that mimic our previously created TM to say if a binary input string is multiple of three or not.

3 Results and discussion

3.1 Exercise 1

As we can see in Figure 2, when using some inputs examples to see the underlying function we see the real function is very close to an $O(n^2)$ function.
If we take a look into our TM logic, what we can see is that it looks continually for a 0 symbol or 1 one. This search is done for the entire string so, our TM iterates over the entire string at least once.
If we analyze the algorithm, we had no best or worst case because the algorithm will do almost the same for every possible combination of our input with a fixed length. Image for a second that our input had many of one of our possible symbols consecutive at the very first and his counterpart following them. If that is the case, the input string will have one type of the symbols until the half of the string and the rest would be the counterpart symbol. Every time the TM find a pair, it will come back to the very first of the string and will look for another pair (if there is one more at) or will iterate the entire string (if no one exists).
With this in mind, my conclusion is that our algorithm has an $O(n^2)$ on average. This conclusion fits perfectly with the results presented in Figure 2.

In the next few lines, i show how i did multiple theoretical functions to compare the projection of the real one against the theoretical ones. The code was programmed in Python using the Numpy library to create and manipulate matrices and Sklearn to create and fit the polynomials.

```
    # Valores reales
    x_real = np.arange(1, 11)
    y_real = np.array([4, 9, 12, 21, 26, 39, 46, 63, 72, 93])

    # Valores predichos para cada polinomio
    x_test = np.linspace(0, 10, num=1000).reshape((-1, 1))
    polyRegressor = PolynomialFeatures((1, 4))      # Cada columna sería X^j
    y_pred = polyRegressor.fit_transform(x_test)[:, 1:]

    plt.plot(x_test, y_pred[:, 0], label='Grado 1')
    plt.plot(x_test, y_pred[:, 1], label='Grado 2')
    plt.plot(x_test, y_pred[:, 2], label='Grado 3')
    plt.plot(x_test, y_pred[:, 3], label='Grado 4')
    plt.plot(x_real, y_real, label='Real')
    plt.xlim(0, 12)
    plt.ylim(0, 100)
    plt.legend()
    plt.savefig('ej1.png')
    plt.show()
```

3.2 Exercise 2

This TM is more simple to analyze compared to the first one. As we can see in the transition table, now we don't have any transition function that modify our readed symbol. In every possible transition function we keep moving to the right until we find the # symbol, where we stop. It's due to not have movements to the left that we won't have nothing worst than an O(n) time complexity function. In fact, if we take a look into the steps the algorithm actually takes to complete his computation for every input example (without taking into account it's length) in Figure 4 we'll see that our hypotheses it's true.
It's because the absence of left and right movements that our algorithm will never take more than n steps to complete his computation.

In the next few lines, i show how i did multiple theoretical functions to compare the projection of the real one against the theoretical ones. The code was programmed in Python using the Numpy library to create and manipulate matrices and Sklearn to create and fit the polynomials.

```
    # Valores reales
    x_real = np.arange(1, 11)
    y_real = np.array([4, 9, 12, 21, 26, 39, 46, 63, 72, 93])

    # Valores predichos para cada polinomio
    x_test = np.linspace(0, 10, num=1000).reshape((-1, 1))
    polyRegressor = PolynomialFeatures((1, 4))      # Cada columna sería X^j
    y_pred = polyRegressor.fit_transform(x_test)[:, 1:]

    plt.plot(x_test, y_pred[:, 0], label='Grado 1')
    plt.plot(x_test, y_pred[:, 1], label='Grado 2')
    plt.plot(x_test, y_pred[:, 2], label='Grado 3')
    plt.plot(x_test, y_pred[:, 3], label='Grado 4')
    plt.plot(x_real, y_real, label='Real')
    plt.xlim(0, 12)
    plt.ylim(0, 100)
    plt.legend()
    plt.savefig('ej1.png')
    plt.show()
```

3.3 Exercise 3

The TM version of the exercise one had to look every time for a pair of 0's and 1's going forward and backward every time until it iterates the entire input or no counterpart exists for a specific symbol. The RAM version is more efficient because it has to iterate just once the entire input string without going backward at any point.
As we can see in Figure 5 the real function is too close to the theoretical linear one. I repeated the experiment for every input length 1000 times to get a closer approximation to the real underlying function. The results are what we expected.

3.4 Exercise 4

If we take a look into our Figure 6, our actual results are close to a theoretical linear function in which the time to process an input increase linearly with respect to the length of the input himself. Although our actual function it's not a perfect linear version, we can appreciate that our function doesn't differs more than a factor of the linear version without going into a polynomial function (like N^2, for example).
This behaviour doesn't surprise us because the program mimic the TM version without doing more operations than the original one. Knowing that it's not hard to conclude that our RAM program it's O(N) like the TM.

4 Conclusions

After been working with TM and programs developed using RAM model what i can conclude is that each model has his advantages and disadvantages. For example, when working with sequential processing, TM could be more easy and intuitive to develop while RAM model it's less intuitive but more close to the way we program high level programs.
What i could tried and i saw by myself it's that a RAM program could be more efficient than TM because of the random access to registers without the need of sequentially iterating over a string.
Finally the answer to the proposed question about the efficiency increase using a RAM program over a TM, as the *Cobham-Edmonds Thesis*, we have to models that are *general and reasonables* so if we develop a program using one of the models, we could resolve the same problem using the other model and the efficiency order would not change more than a polynomial factor. As we saw in the results of exercise 1 and 3, the first one used a TM while the second used a RAM program and the factor was polynomial (in this case, N). Exercises 2 and 4 doesn't change too much because we used a similar approach to resolve them.

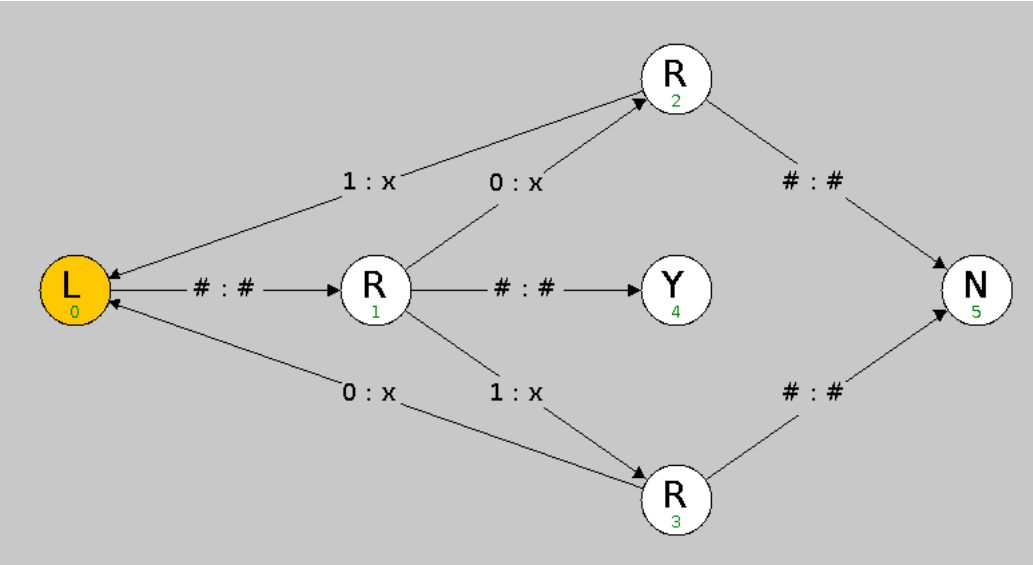

Figure 1: Turing Machine that recognizes if an input string has the same number of zeros and ones.

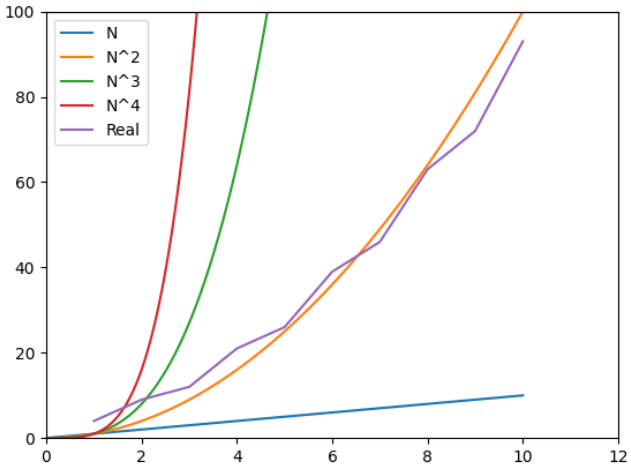

Figure 2: Comparison of real processing vs theoretical functions facing number of steps taken to process the input string against the input length.

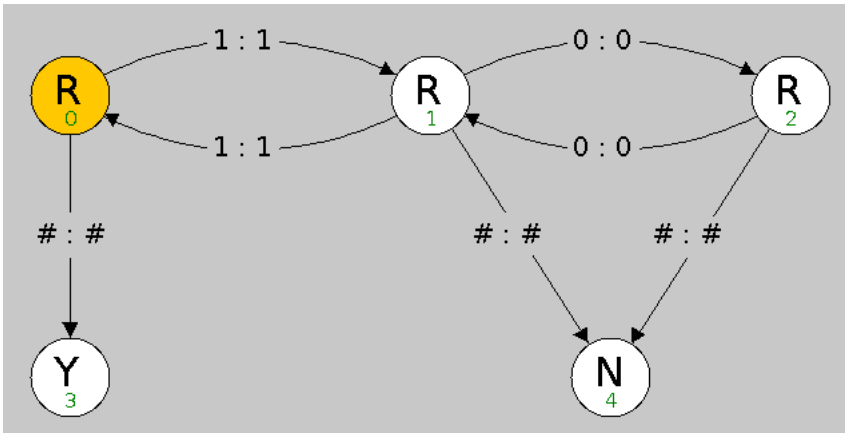

Figure 3: Turing Machine that recognizes if an input string is multiple of three.

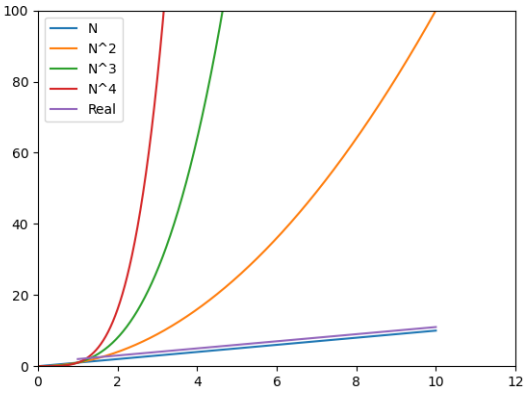

Figure 4: Comparison of real processing vs theoretical functions facing number of steps taken to process the input string against the input length.

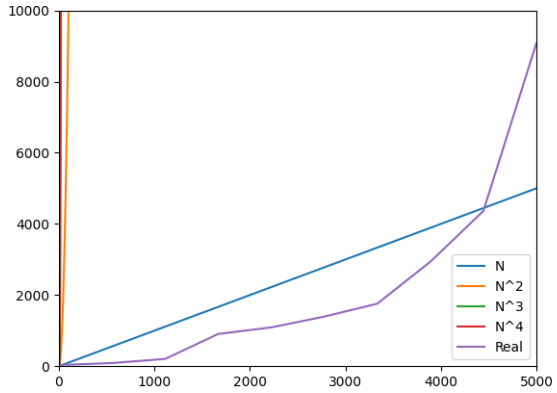

Figure 5: Comparison of real processing vs theoretical functions facing time taken to process the input string against the input length.

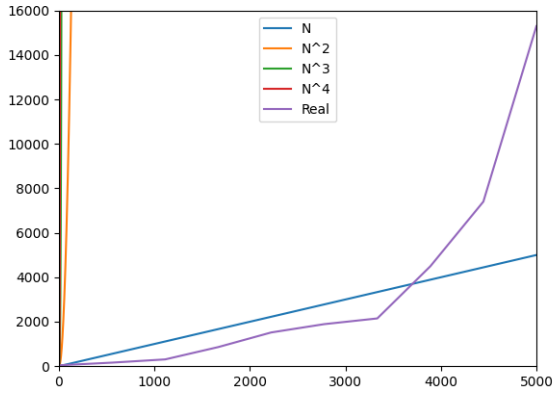

Figure 6: Comparison of real processing vs theoretical functions facing time taken to process the input string against the input length.

5 References

[1] Robert Sedgewick and Kevin Wayne. 2017.
URL: https://introcs.cs.princeton.edu/java/52turing/.

Bibliography

[1] Stephen A. Cook and Robert A. Reckhow. *Time Bounded Random Access Machines*. 1973.
URL: http://www.cs.utoronto.ca/~sacook/homepage/rams.pdf.

[2] Oded Goldreich. *P, NP, and NP-completeness. The basics of computational complexity*. Cambridge University Press, 2010.

[3] Robert Sedgewick and Kevin Wayne. 2017.
URL: https://introcs.cs.princeton.edu/java/52turing/.