CS3052 Practical1 Report

160001055

March 2019

1 Introduction

This practical consisted in building a Turing Machine simulator, capable of parsing a representation, and then, given a tape, executing until either the machine accepts or reject. In addition to this, we had to show proficiency in solving two given problems and two chosen ones. In particular, I implemented a machine which outputs an exact binary representation of $\sqrt{2}$ and a machine which executes a simplified dialect of "Brainfuck"[1] [1]. Finally, as an extension, I implemented a NDTM and solved the repeated words problem as requested.

2 Simulator

The simulator is the core of the practical. It is supposed to parse an input file, construct a Turing Machine [2] from it and then run it until acceptance or rejection. For the sake of this practical, I have decided to write the simulator in Rust. The choice of Rust is not really dictated by necessity, but rather by my general liking of the language, with both its performance and memory benefit and the relatively high abstraction level it provides.

2.1 API

The Simulator builds a single executable, which takes arguments in the following form: `main [FLAGS] [OPTIONS] <FILE> [TAPE_FILE]`.
In particular `FLAGS` can be either `--help`, `--version` or `--nondeterministic`. In particular, the latter specifies if the machine should be a **NDTM**.
`OPTIONS` can be any of `--limit` or `--tape`. The former specifies a limit on the number of computations the machine can take before rejecting (default: ∞), the latter specifies an inline tape (default: `""`).
`FILE` is a path to the file where the TM description is located
`TAPE_FILE` is a optional path to a file where the tape to be used is located.
A more detailed description is available using the `--help` command, which also shows all the abbreviations available with the above flags and options.

[1] While some publications use Brainf**k or similar censoring, I believe that, in the spirit of the language features, it is necessary to use its given name

All of this was generated using the **clap** [3] library in Rust, which takes care of arguments order and conflicts to allow for a robust interface.
Furthermore, the user can specify the verbosity of the program by setting the `RUST_INFO` environment variable to either `info` or `debug`.

2.2 Design

The design for this application is, admittedly, a bit more involved then what it could have been. The core idea is that a Turing Machine consists of three fundamental components: the `MachineRepresentation`, the `TransitionTable` and the `Tape`. Each of these components need to be validated on its own, and furthermore they need to be valid when composed. Furthermore, this validation might (and tendentially will) be different with different kinds of machine (e.g. **TM, NDTM,** k**-TM**). To solve this problem, I have created a trait hierarchy, with an associated builder hierarchy. In particular, the traits are:

- `TuringMachine`
- `MachineRepresentation`
- `TransitionTable`

. The builders are:

- `TuringMachineBuilder`,
- `MachineRepresentationBuilder`
- `TransitionTableBuilder`

. The idea is that the builder will expose an extremely general interface that allows for setting the values (i.e. from a parser) and initial validation. Then each implementor of the corresponding trait can use such builder to initialize, doing validation in the meantime. To validate the ensemble, we use a transitive approach.
So the `MachineRepresentationBuilder` also exposes a corresponding `TransitionTableBuilder`, so the `MachineRepresentation` can use that to validate the pair. Then the `TuringMachineBuilder` contains both a `MachineRepresentation` and a tape, so when it is used to build a `TuringMachine` it can do the correct validation of the two.
The following subsections are meant to give some insight on the hierarchy, but I believe the reader might find the generated **rustdoc** more insightful and easy to navigate. In that case, I suggest to run `cargo doc` in the project directory, and then navigate to (`target/doc/turing_machine/index.html`)

2.2.1 TuringMachine

A trait which is implemented for types that simulate a **TM**. It exposes 6 methods, most notably `step` and `from_builder`. The TM T in question will simulate

the abstract one :

$$(Q, \Gamma, q_0, q_a, q_r, \delta) \subseteq \texttt{T::ReprTy}$$

where Q is the set of states, with $Q \subseteq \texttt{T::StateTy}$, Γ is both the input and tape alphabet, and $q_0, q_a, q_r \in Q$ are respectively the start, accept, reject state. Finally $\delta \in \texttt{T::ReprTy::TableTy}$ is the transition table function[2].

- `step`, if the machine is not in a accept or reject state, it will perform on step of calculation. In a **TM** this is equivalent to applying the transition function once.
- `is_accepting, is_rejecting`, check the machine has finished executing
- `run`, run until one of the above states is reached.
- `build_from`, given a `TuringMachineBuilder` with the correct signature, construct a **TM**

2.2.2 MachineRepresentation

This trait is a immutable representation of a **TM**. The implementors T of `TuringMachine` usually use a implementor R of `MachineRepresentation<T::StateTy>` to apply transition table, get list of states and similar. As above, this trait represents the tuple:

$$(Q, \Gamma, q_0, q_a, q_r, \delta)$$

such that Q, q_0, q_a, q_r satisfy the above conditions, $\Gamma \subseteq \texttt{R::InputTy}$ and $\delta \in \texttt{R::TableTy}$. All of its methods apart from `from_builder` are self explanatory accessor methods, while the former construct a representation from a `MachineRepresentationBuilder` which exposes matching associated types.

2.2.3 TransitionTable

This trait is possibly the most fundamental one of them all, as this component is the one the most radically changes when we shift from **TM** from **NDTM**. The trait has three associated types, `StateTy, InputTy, OutputTy`. It represents the set of all functions δ with signature $\delta : (\texttt{StateTy} \times \texttt{InputTy}) \to \texttt{OutputTy}$. For example, for a **TM** we have $\texttt{InputTy} = \texttt{char}$ and $\texttt{OutputTy} = \texttt{Action<StateTy>} = \texttt{StateTy} \times \texttt{char} \times \{L, R, S\}$. For a **NDTM** all is similar but $\texttt{OutputTy} = 2^{\texttt{Action<StateTy>}} = 2^{\texttt{StateTy} \times \texttt{char} \times \{L,R,S\}}$ where by 2^X we denote the power-set of X.

2.2.4 TuringMachineBuilder

Following the rust builder pattern [4], this type allows you to set a representation and a tape, and can be used to build a `TuringMachine`. It does validation checking that the representation's alphabet and the tape alphabet correspond.

[2] In these sections, I have been using the `Type` to denote the set of all elements of type `Type`. I am not sure if it is abuse of notation, but I felt it was important to specify. Example: $\texttt{usize} = \{0, 1, ..., \texttt{usize::MAX}\}$ and $\texttt{String} = \texttt{char}^* = \texttt{Vec<char>}$

2.2.5 MachineRepresentationBuilder

This trait is implemented by types that can be used to build a `Machine Representation`. It exposes setters and getters methods, the setters are supposed to be used to add information to the builder (e.g. new states, alphabet symbol), while doing individual validation. The getters are then used by the `from_builder` method of the representation to construct it.

2.2.6 TransitionTableBuilder

This trait is implemented by types that are meant to be used from a `TransitionTable`'s `from_builder`. Differently from the `MachineRepresentationBuilder` it works by parsing a line of text at a time. It then has various getters that can be used for validation and for the actual building.

2.3 Implementation

The implementation of the simulator, after being given the trait hierarchy, is extremely straight forward. I will be giving here some small details.

- The parser is essentially a line-by-line parser, which tokenizes by whitespace, and propagates errors up the call stack using the `Result` type. Whenever a line parse is successful, the information gathered is then stored in a builder.
- Each builder contains one or more `HashSet, HashMap` mappings which are used to store the configuration. This allow for efficiently detecting duplicate states, alphabet ecc.
- When each builder is converted to a trait object, further validation is conducted (this is in addition to the strong type-checks that the design provides). This is usually implemented using set difference operations.
- Each type of **TM** exposes a type which implements `TransitionTable` which is usually implemented by a `HashMap` lookup.
- To implement the step counting and limiting, instead of littering each **TM** with a step counter, I have implemented two extension structs, which wrap a `TuringMachine` with the counting and limiting algorithm.
- Instead of using `std::collections::{HashMap, HashSet}`, I have used `hashbrown::{HashMap, HashSet}`. This is because the `hashbrown` implementation uses Google's Swisstable algorithm [5], which has noticeable performance benefits. This provided notable speedups, in particular on the `busy5` test, which had a 40% speedup.

3 Bracket Matching

The first problem that I had to solve was the creation of a **TM** that recognizes the language of balanced brackets. First of all, I have noticed that this is a context free language, generated by:

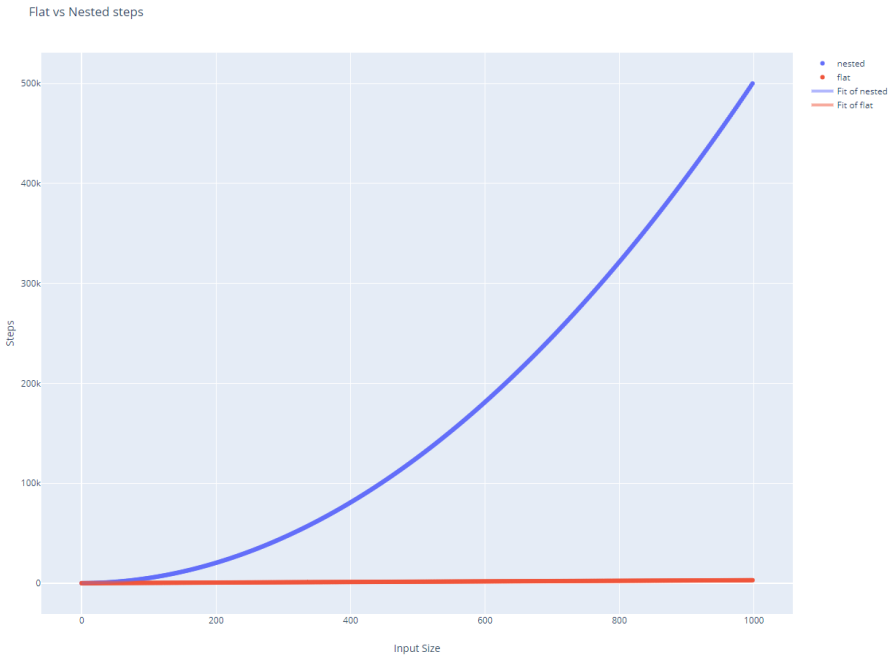
$$S \to (S)|SS|\epsilon$$

While this is not fundamentally important, it implies that such a machine exists. The algorithm works as follows:

1. If $w = \epsilon$, accept. Else, if the first cell is (mark it as $\wedge$ and move right
2. If the current cell is) replace it with \$, move left and go to 3. If it is a blank, move left and go to 4. Else move right and repeat.
3. If the current cell is (replace it with \$, move right and go to 2. If it is $\wedge$ replace it with [, and go to 2. Else go left and repeat
4. Scan left, rejecting if the symbol is not \$ or [. Accept if [.

To provide an example, this is how the string ()() would be transformed: $\wedge$)(), $\wedge$\$(), [\$(\$, [\$\$\$. Since the last string is composed only of \$ and [, it would be accepted. Another string, such as (()) will be transformed in this way: $\wedge$()), $\wedge$(\$), $\wedge$\$\$), $\wedge$\$\$\$, [\$\$\$ and then be accepted. There are some interesting points to be extrapolated from this. First of all, a string of size n that is to be accepted will be transformed to one of the form [\$$^{n-1}$ and this will be verified in exactly n steps. From this we can conclude that this algorithm is $\Omega(n)$. Also, from counting the number of steps, we can see that nested parenthesis need more operation to be solved when compared to flat ones. In particular, we can show that the number of steps that the machine on an input w of size n where the $w = ()^{n/2}$ takes exactly $3n$ steps [3]. On the more interesting case $w = (^{n/2})^{n/2}$ instead the number of steps follows a quadratic function, in particular $\frac{n^2}{2} + 2n$. So, taking worst case scenario, this algorithm is $O(n^2)$. This is not the optimal complexity, as a simple stack algorithm can be shown to be $O(n)$, but it is definitely simple enough for this practical. In particular, this machine executes the 500k required for a string of length 1000 in 750ms when built without optimizations, and in 58ms when optimized.

[3] Note, when I say that $w = ()^{n/2}$ mean that the string "()" is repeated $n/2$ times, I just could not find a notationally elegant way to show that. Also, if n is not even then the string can trivially be rejected



4 Binary Addition

The second language that I had recognize on the machine is $L = \{w_1\#w_2\#w_3|w_1 + w_2 = w_3\}$, where w_1, w_2, w_3 are numbers in binary with the leftmost digit being the least significant. The algorithm that I designed for this is pretty simple in concept, but because of the limitations of the **TM** it proves to be pretty hard to write and understand. In essence the algorithm is as follows, we store the values of a, b, c by using one state for each combination (8 in total):

1. If the current char is #, shift everything one position right. Print $\wedge$ to mark the start position. Move right.
2. $c = 0$
3. Skip right to the first non$ character.
4. if the current char is 0 or # then set $a = 0$, else if it is 1, $a = 1$. Replace the current symbol (if it is not #) with $. Stay in place.
5. Move right to the next # symbol.
6. Move right to the first non$ symbol.
7. If it is 0 or # then $b = 0$, if it is 1 then $b = 1$. Replace the current symbol with $ if it is not #. Stay in place.

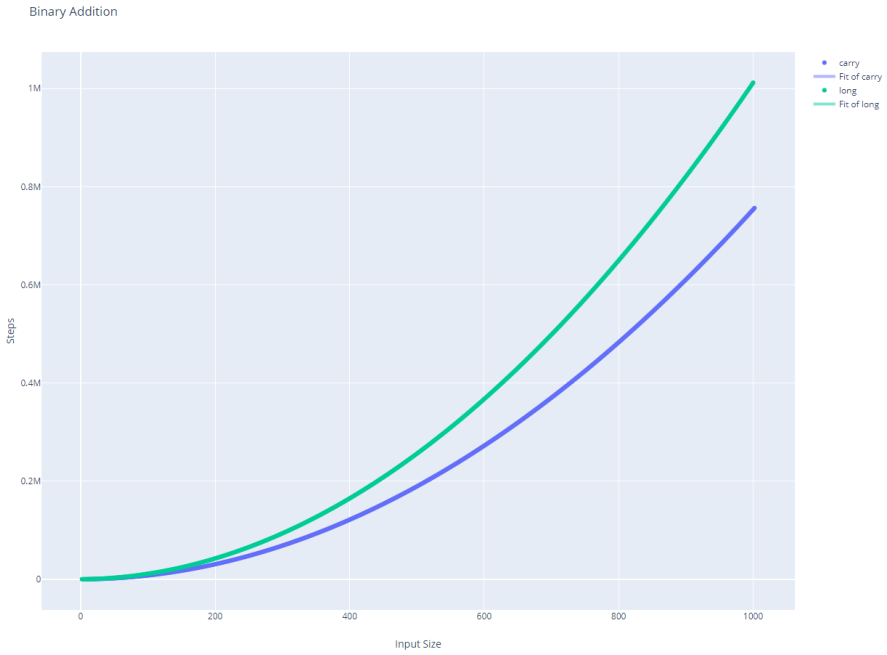
8. Move right to the next # character.

9. Skip to the next non$ symbol.

10. Read the symbol to r, if it is a blank interpret it as 0. If a + b + c = r write $, go to start and then run 3, with c if $a + b + c > 1$. Else reject. If $a = b = c = 0$ and the current symbol is a blank then go to 11.

11. Check that the tape contains only ∧, #, $. If it does, accept. Else reject.

As before, here is an example for the string 0#0#0: ∧#0#0, ∧#$#0, ∧#$#$, A more complex one (2 + 2 = 4) 01#01#001: ∧1#01#001, ∧1#$1#001, ∧1#$1#$01, ∧$#$1#$01, ∧$#$$#$01, ∧$#$$#$$1, ∧$#$$#$$$
Analysing this function is a bit more challenging, as there are no two clear variants that the problem can come in. However, we can make some estimates. Consider a string $\{0,1\}^a\#\{0,1\}b\#\{0,1\}^c$. On the first pass the algorithm takes $a + b + 2 + 1$ steps, on the second $a + b + 2 + 2$ and so on until on the last one which takes $a + b + c + 2$ steps. So in total we will have $c(a+b+2)+\sum_{i=0}^{c} i = a*c+b*c+2*c+\frac{c(c+1)}{2} = \frac{c}{2}(2a+2b+c+5) = \frac{c}{2}(a+b+3+n)$. While this is a very rough estimates (for example it does not take into account the backtracking to start), it does suggest that this algorithm is quadratic in nature (since in the average case usually $a \approx b \approx c \approx n/3$). I have then taken two particular case, the string $1^x\#1\#0^x1$ for $x \in \mathbb{N}, n = 2x + 4$ and the family $\#\#0^{n-2}$. As the previous analysis shows, the nature of the passes implies that the complexity of the algorithm grows with the square of c while it only grows linearly with the length of $a+b$. The two cases selected have respectively $c = \frac{n}{2} - 3$ and $c = n - 2$. So they should accurately represent a average and worst case scenario. As the graph and the fit show, we can fit $0.75x^2+3.5x+1.3$ on the first sample, $x^2 + 12x - 11$ on the second one, both with $R^2 = 1$. So we can reasonably conclude that this algorithm is $O(n^2)$.



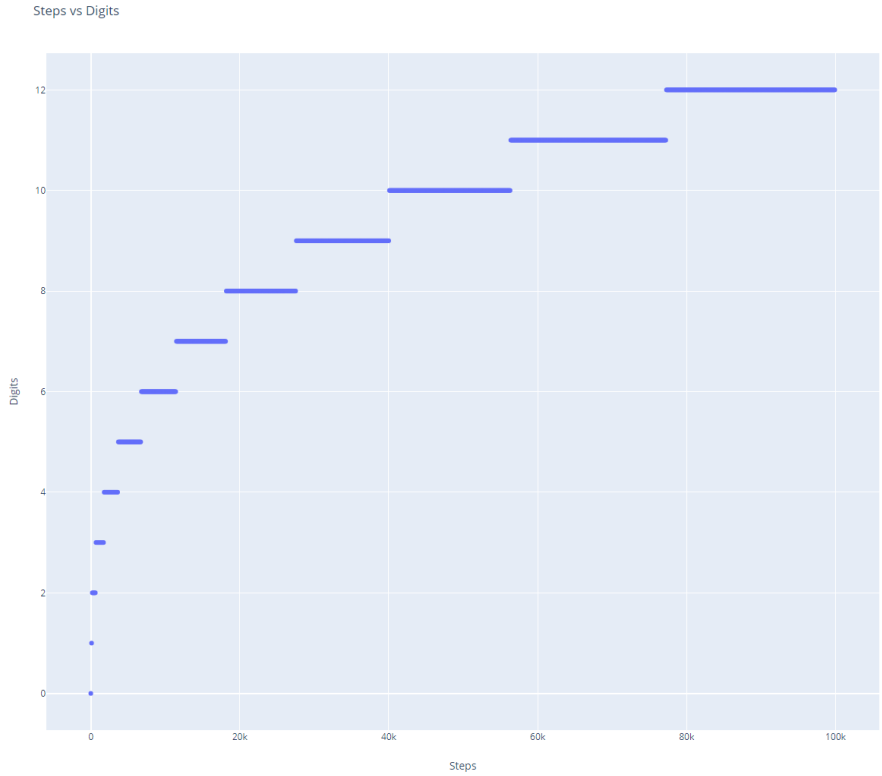
5 Square root of 2

This Turing Machine is based over the one described by Charles Petzold in his book "The Annotated Turing" [6]. This machine computes the binary representation of $\sqrt{2} = \sqrt{10_2} = 1.01101010000010011110..._2$ in the oddly numbered squares. I would note that this machine makes use of an extension to the **TM** language, which enables to have S transitions in addition to $\{L, R\}$, so this machine will not be runnable using `reftm`. In contrast with the other machines that I have described, this machine does not terminate, it simply continues computing digits after digit until the end of time. For this reason, I strongly suggest to either test it using the debugging mode (enabled by setting the environment variable `RUST_LOG` = `debug`) or by limiting the number of steps (using `--limit`). The machine starts by assuming the first digit is 1, it act recursively according to this algorithm:

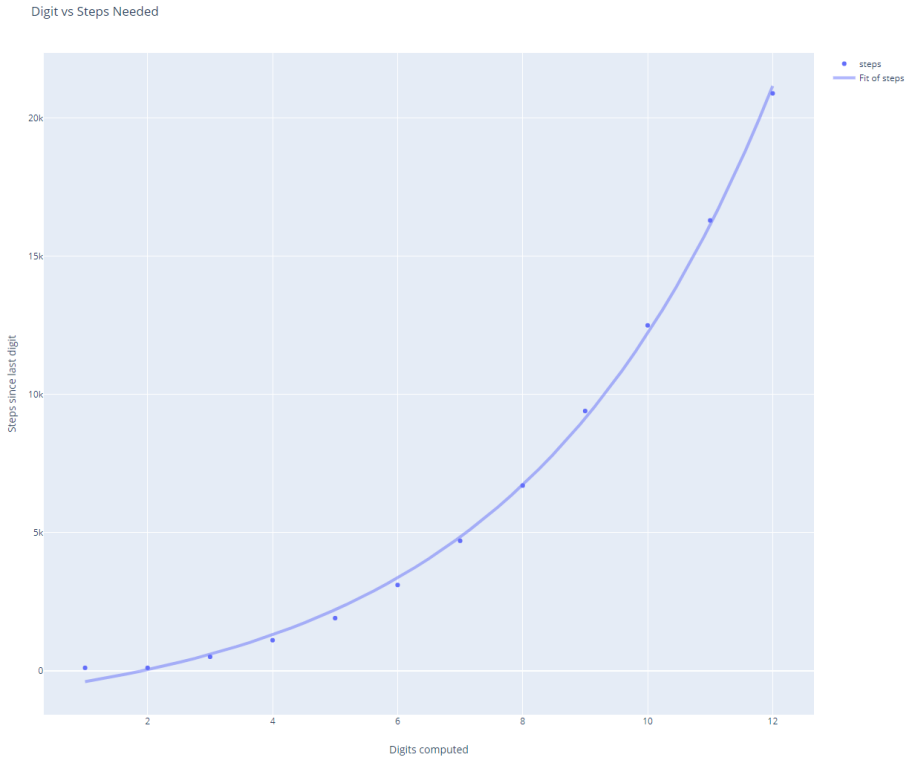
- Multiply the n digits we have so far by themselves
- If the result has $2n - 1$ digits we guessed correctly, write 1
- If the result has $2n$ digits we are over 2, so write 0

The multiplication itself is done on the evenly numbered squares of the machine, which implies that every time a digit is printed it is definitively correct. Furthermore, this result is an exact computation, so given enough time the machine will compute $\sqrt{2}$ to any desired level of precision (as long as we have enough

physical memory of course).
While complexity analysis is not possible to do in a traditional fashion, as the function never terminates, we can estimate the time needed to print the n-th digit.



As the graphics show, the time to compute the n-th digit is constantly increasing. In particular, we can compute the number of extra steps that are need to compute the n-th digit from the $(n-1)$-th.



Now, by numerical investigation one can notice that $\frac{\Delta steps(n)}{\Delta steps(n-1)} \approx 1.3 \implies \Delta steps(n) \approx 1.3 \Delta steps(n-1)$. This recurrence relation, can be solved then as $\Delta steps(n) \approx 1.3^{n-1} \Delta steps(1)$, which seem to suggest that the algorithm experiences a exponential growth, which in particular would be $O(1.3^n)$ for computing the n-th digit when the $(n-1)$-th is known [4].

6 Brainfuck

Finally, as my last trick in the bag, I have implemented a interpreter for a subset of the programming language "Brainfuck" [1]. First of all, I believe an introduction is needed. The Brainfuck programming language is an esoteric programming language, which, despite having an extremely restricted set of operation at its disposal, was proven to be Turing Complete. A Brainfuck program consists of the following symbols:

[4] Note, this is a very rough estimate, developed from a small sample of data points, which, while I believe it is valuable, should be taken with a grain of salt

symb	action
+	increment cell
-	decrement cell
<	move pointer left
>	move pointer right
[	if cell==0, skip to]
]	if cell==0, rewind to [
,	read one byte from stdin to cell
.	print cell's byte to stdout

In particular, my version restricts itself to $+, -, <, >, [,]$. My version, which from here upon I will call RBrainfuck, also does not support nested brackets, and instead of having each cell value be a byte in size, restricts it to a bit. The **TM** that I have designed accepts if the RBrainfuck program is valid and terminates. The algorithm that I have devised to do so is as follows.
First of all, given a string s, we format the tape to be @\$$s$#$\wedge$, in particular, the @ character marks the start of the tape, \$ marks the next instruction to execute, # marks the start of the program's tape, $\wedge$ marks the position of the pointer. The algorithm then works as follows:

EXECUTE

1. Move to \$, and step right
2. Read the character, and select the correct action based on the character.

On + (−):

1. Move right until $\wedge$, move right
2. Write 1 (0) on the cell
3. Move left until \$
4. Go to **MOVE_FORWARD**

On <, (>):

1. Move right until $\wedge$, move left (right)
2. If the current char is # do nothing (the tape is not doubly infinite), else write $\wedge$ and write the current char to the right (left).
3. Move left until \$
4. Go to **MOVE_FORWARD**

On [:

1. Move right until $\wedge$, move right
2. Read the char to c
3. Move left until \$
4. If $c = 1$, go to **MOVE_FORWARD**
5. Go to the next], write \$, set $s =]$
6. Move left until \$, writing s on the char and setting s to be the most recently read char
7. Go to **EXECUTE**

On]:

1. Move right until $\wedge$, move right
2. Read the char to c
3. Move left until \$
4. If $c = 1$, go to **MOVE_FORWARD**
5. Go left to next [, write \$, set $s = [$
6. Go right until \$, writing s on the char and setting s to be the most recently read char
7. Go to **EXECUTE**

MOVE_FORWARD

1. Swap current char (\$) with the one its right
2. Go to **EXECUTE**

For this kind of machine, a length based complexity analysis is certainly hard, and I believe that the complexity would be in the order of the busy beaver function [5].

[5] That is, if this subset of Brainfuck is Turing Complete, otherwise I am not sure. The busy beaver function $\Sigma(n)$ is the number of ones printed by the TM of n states that terminates and that writes the most one. It is an uncomputable function that grows quicker than any other computable function.

7 Non determinism

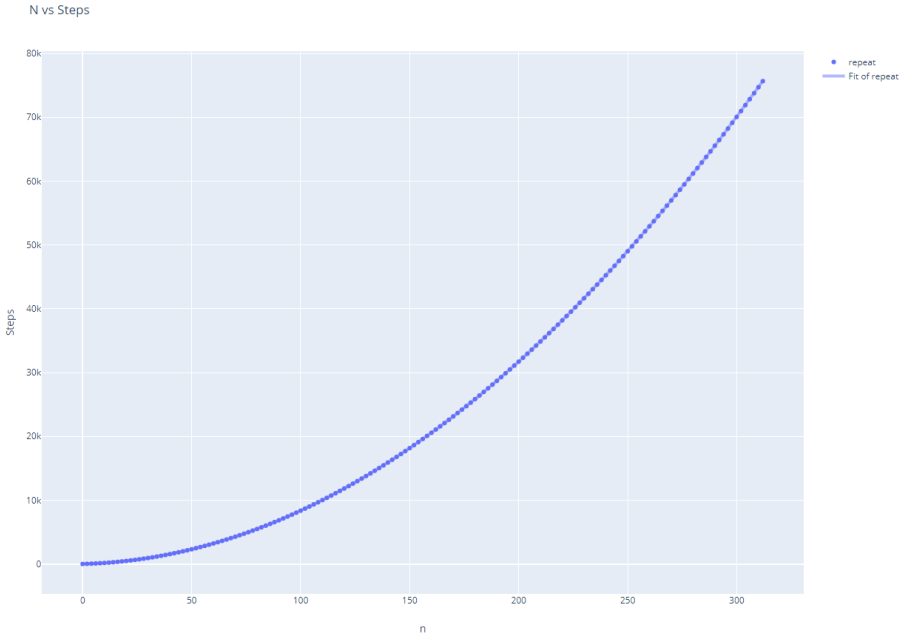
Furthermore, I have implemented non determinism pretty easily. Thanks to the design I managed to use all the same machinery as for the deterministic TM, and simply relaxing some checks for correctness. The **NDTM** implementation essentially maintains a list of tuples of $(state, pos, tape)$, and on each transition which use non determinism it copies the corresponding tuple and apply the various different transitions. The machine accepts if one of the states in the list of tuples is an accept state, and it rejects if all of such states are reject states.

8 Repeated Words

The non deterministic problem to solve was to create a machine that is able to find whether a binary input string is the concatenation of two equal sub-words. The approach that I developed is deceivingly simple. First of all, we mark that tape with a $\wedge$ symbol. Then we read the tape from right to left, and on each read symbol $\{0, 1\}$ the computation splits. One path will continue the reading, while the other will transform the tape as follows: if the tape is $\wedge s_1 s_2$, where the cursor is on the first symbol of s_2, it will modify it to read $\wedge s_1 \# s_2$. Then, the path of computation that transformed the tape will go to the start of the tape, and it will execute the following algorithm:

1. Go to start of tape
2. Move right, skipping \$ characters until the first binary digit is found (If # is found go to 4). Name this digit a, and write \$ in its place.
3. Move right until #, then move right skipping \$ until first binary digit. Compare this digit with a, if they are not the same reject, else write \$ in its place. Move to 1.
4. Move to end of the tape, checking no binary character are present. If we reach a blank accept.

The complexity of this algorithm is harder to analyze, especially since the much bigger weight of a step makes gathering a large data-set harder. However, some interesting facts can be found. First of all, the number of steps are dependent only on the number of input character rather than the content of the tape.



Secondly, the diagram is extremely similar to the one for nested brackets, and it suggests a quadratic relationship. In particular according to my trend-line fitting the relationship has a $R^2 = 1$, strongly suggesting this behaviour. So we can conclude this algorithm is $O(n^2)$.

9 Conclusion

In conclusion, I found extremely enjoyable to work on this practical, especially because it goes to touch a sector of computer science which is lesser know and extremely challenging. In particular, I truly enjoyed the writing of the simulator, especially for the freedom that it allowed in selecting the programming language of your choice. Being very much a rust aficionado, the ability to use it in a university practice made it both easier and more interesting! Finally, the problem solving aspect was pretty insightful, as it truly shows the intrinsic power of such machine (and the enourmous amount of peril that can come from them!).

References

[1] U Müller. Brainfuck–an eight-instruction turing-complete programming language (1993).

[2] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[3] clap rs. A full featured, fast command line argument parser for rust. `https://clap.rs/`.

[4] Rust WG. The builder pattern. `https://doc.rust-lang.org/1.0.0/style/ownership/builders.html`.

[5] Kulukundis Perepelitsa Benzaquen, Evlogimenos. Swiss tables and absl::hash. `https://abseil.io/blog/20180927-swisstables`.

[6] Charles Petzold. *The annotated Turing: a guided tour through Alan Turing's historic paper on computability and the Turing machine*. Wiley Publishing, 2008.