CS3052

150001088

27 April 2018

Contents

Overview **1**
Reproducability . . . 1

Design **1**
Implementation . . . 1
Complexity Testing . . . 2

The Problems **3**
Palindrome . . . 3
Binary addition . . . 6
Divisibility by 3 . . . 11
Busy beaver . . . 12
Comparison of the Problems . . . 16
Summary . . . 16

Non deterministic TM **17**
Design . . . 17
Binary word . . . 17

Conclusion **20**

Bibliography **20**

Appendix **21**

Overview

This practical offers an example of a Java programmed Turing Machine (TM) and encodes four different TM problems into a human readable and machine readable format. A theoretical and experimental analysis of complexity is executed on the problem of Palindrome recognition, Binary addition, Divisibility by three and 2 state busy beavers. In the last chapter a non deterministic Turing Machine simulator is implemented.

Reproducability

All my experiments are reproducible with the Code submitted. All relevant data files are created when the main method is run. The main method can be run from the command using standard command line options. Further, the code fulfills the requirement such that *java Main input_file input_string type* will test the input string against the given encoded requirements in the input file. *type* can be "NTM" for a non deterministic TM, "Test" generates the data files for the problems below and any other input will run a deterministic TM. However, *type* should be given in any way otherwise the program will fail. This report has been created with R markdown and the plots in this project have been made with different R libraries. Relevant R code chunks are included in this report.

Design

This report has been planned with a reader in mind to give an introduction in different Turing Machine problems and their theoretical and experimental analysis. The Java code to conduct the experiments is split up into a easily and digestive format and should not be the worry in the process to understand the implications of this report. The experiments were designed to cover growth sizes in the algorithm to be able to derive a general feeling for the growth factor, but still small enough to easily reproduce the dataset without a bigger time investment. For every problem a file with the machine readable description of the program is included in this submission. The failes carry the name of the test they will be included in.

A test class has been provided with the name "Test_TuringMachineTest.java" and uses Junit Jupiter annotations.

Implementation

This implementation in Java offers an object oriented approach to Turing Machine encoding. A TM is considered an object and created by a Turing Machine Factory, which takes a TM description in a format specified by the practical specifications. This factory builds a Turing Machine by reading the input file and adding states and transitions to a Turing Machine object. A transition is a new transition object, which contains the read state, read symbol, write state and write symbol as well as the move direction.
The TM object itself has multiple fields including HashSet's for all states, the accepting states and the transition objects. Further, the start state is specified and three different fields maintain information while running: the tape, the current state and the current symbol. The TM object can then be used to run a check on an input string. A verbose option is included by setting steps to **true**. Upon success the run function returns an integer value greater or equal to zero, indicating the count of transitions. If the input string is rejected, -1 is returned.

Note that this implementation does *not* check if the states of an added transition nor symbols were specified before in the states or the alphabet. This has been discussed with the lecturer before submission and was left to the implementation.

Complexity Testing

To allow easy testing a wrapper class has been written around the Turing Machine and the Turing Machine Factory. The TimeTaker class has a function TuringTimer, which takes the start input size and end input size of the problem, the name of a text file containing the description of the Turing Machine and the output file name for the results. The name TuringTimer can be considered slightly misleading as transitions are counted instead of time, but these transitions can be considered as another type of time measurement as it counts the operations needed to compute the problem and as time is machine specific, counting transitions offers a more generalised description of the problem complexity. Internally the TuringTimer function creates a TM object as described in the previous chapter and increases the problem size step by step. Problem size should not be confused with the input length as they can differ depending on how the problem size is defined. Taking binary addition as an example a problem size of 1 refers to a problem in which the integers added have size one, e.g. "0#0#0". We see that the string has size 5.

The creation of problems is randomised with seed 1234 and fitted to the input file name. The file descriptor is analysed and the problem of the specified size is returned, all returned problems will be accepted ultimately. There is no cleaning of rejected inputs. The output file contains the input which was analysed, the transitions and the problem size.

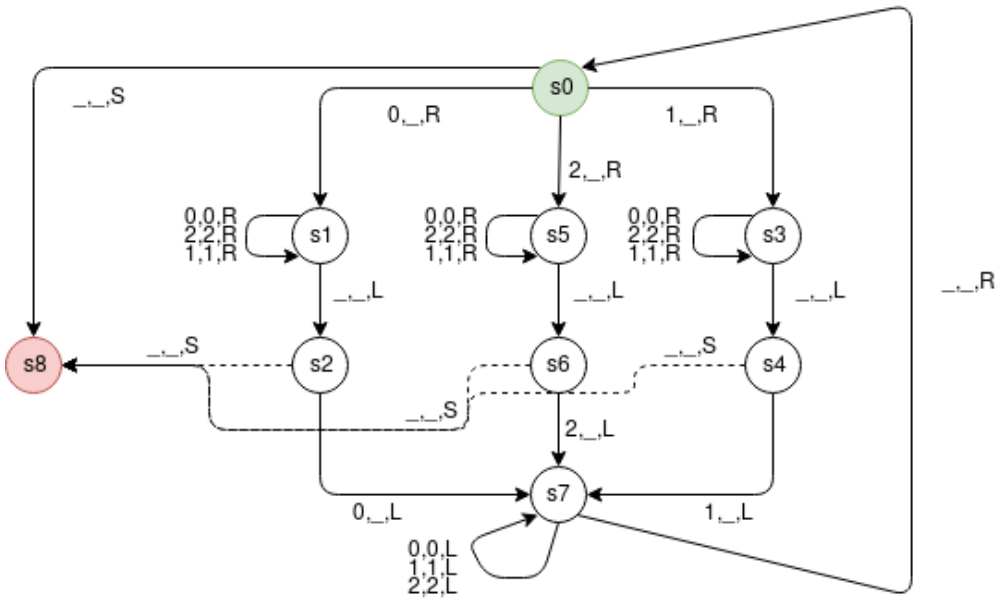
The Problems

In the subsequent sections 4 problems will be analysed. For three of the problems a unittest class is implemented to prove the validity of the developed TM. For the last problems references are given to proof the correctness. The four problems are:

1. Palindrome
2. Binary Addition
3. Divisibility by three
4. Busy Beaver 2 state Turing Machines

Palindrome

A palindrome is an input string which is equivalent to its reversed version. Hence madam is a palindrome, but Adam is not. For this analysis the alphabet is restriced to {0,1,2}. The solution has been visualised below in a graph:



The state marked in red describes the accepting state, green is the start state.

Given an TM = (Q,Σ,I,q,δ,F), here we have:

Q = {s0,s1,s2,s3,s4,s5,s6,s7,s8}
Σ = {I,_}
I = {0,1,2}
q = {s0}
δ = as defined in the diagram
F = {s8}

The file describing the very same version encoded in a machine readable format is named “palindrome.txt” and included in this submission.

The input size is defined to be the amount of symbol pairs. This has been done to easily create palindromes. These palindromes will always be even, however, his has no greater impact on the validity of the analysis or the implementation.

Validity of implementation

The validity of this implementation has been tested with a java unittest suite. Beyond the examples given in the specification, this test suite includes more sophisticated corner cases, including uneven Palindromes which consists of the same input symbol. Further, this example has been tested by automatically generated inputs. The automatic generation of a palindrome appends one of {0,1,2} to the beginning and end of the input string. For various seeds, this implementation has proved to detect correct and incorrect palindromes accurately. This implementation could have been improved if there was a two way check implemented. This means that "on the way back" from one end the Turing machine checks again, instead of traversing the whole string before picking up new information about the outermost character.

Theoretical complexity analysis

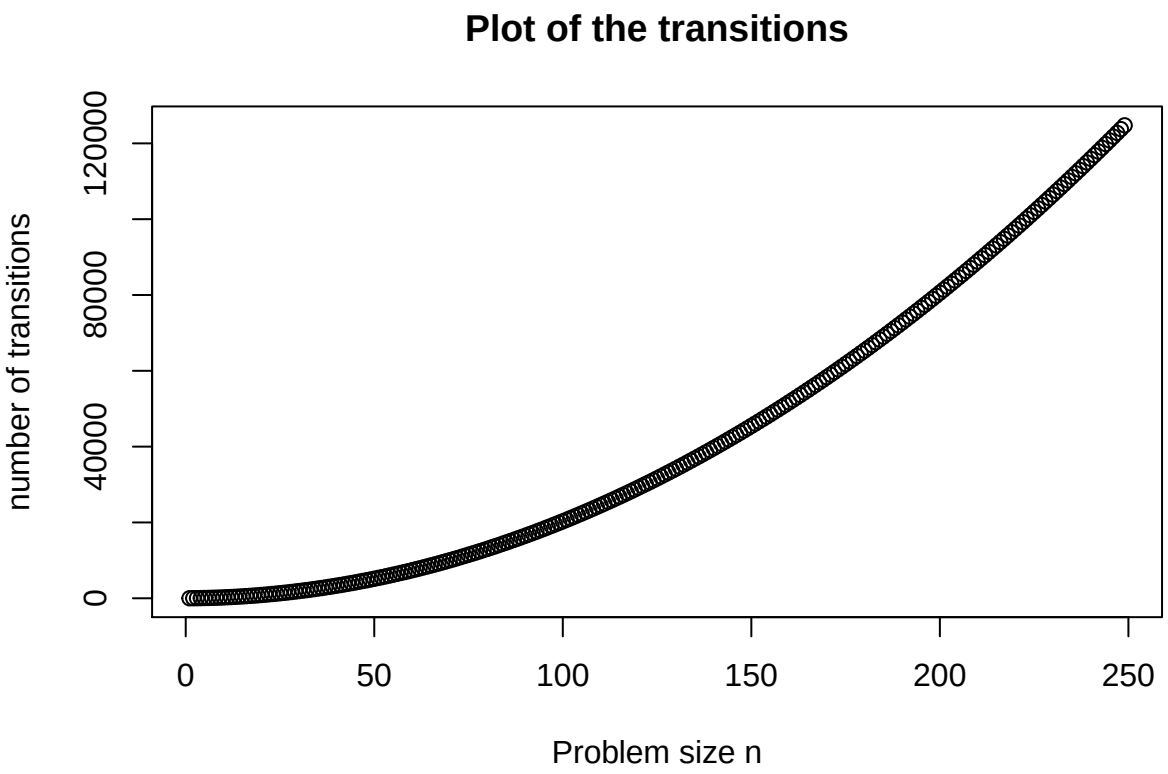
With every single correct input, there will always be at least one transition. This is also true for the empty input. This is because the state needs to change at least ones into the accepting state. For an input size of n, the algorithm will read the first input and then traverse until the head points to an empty symbol and then the head will traverse back to the start of the string. This means that during the first traversal, there will be n+1 transitions in total. Traversing back will only require n transitions as the initial symbol has been removed from the string. Hence, there will be 1 less transition. This will continue until the input string has been reduced to size 0 and only the empty symbol is left on the tape. Writing out the above described process yields:

$$N = 1 + (n+1) + n + (n-1) + ... + 1 = 1 + \sum_{i=1}^{n} (n+1-i) \leq n + n + ... + n$$

From this, we can conclude the inequality $N \leq n^2$ is true and that the palindrome TM is upper bounded by $O(n^2)$.

Experimental complexity analysis

As defined above, input size of 1, will actually be a string of length 2. E.g. the string "22" is input size one. This is down to the creation when the size parameter attaches two extra characters to the front and end of the string. The sampled data is displayed below:

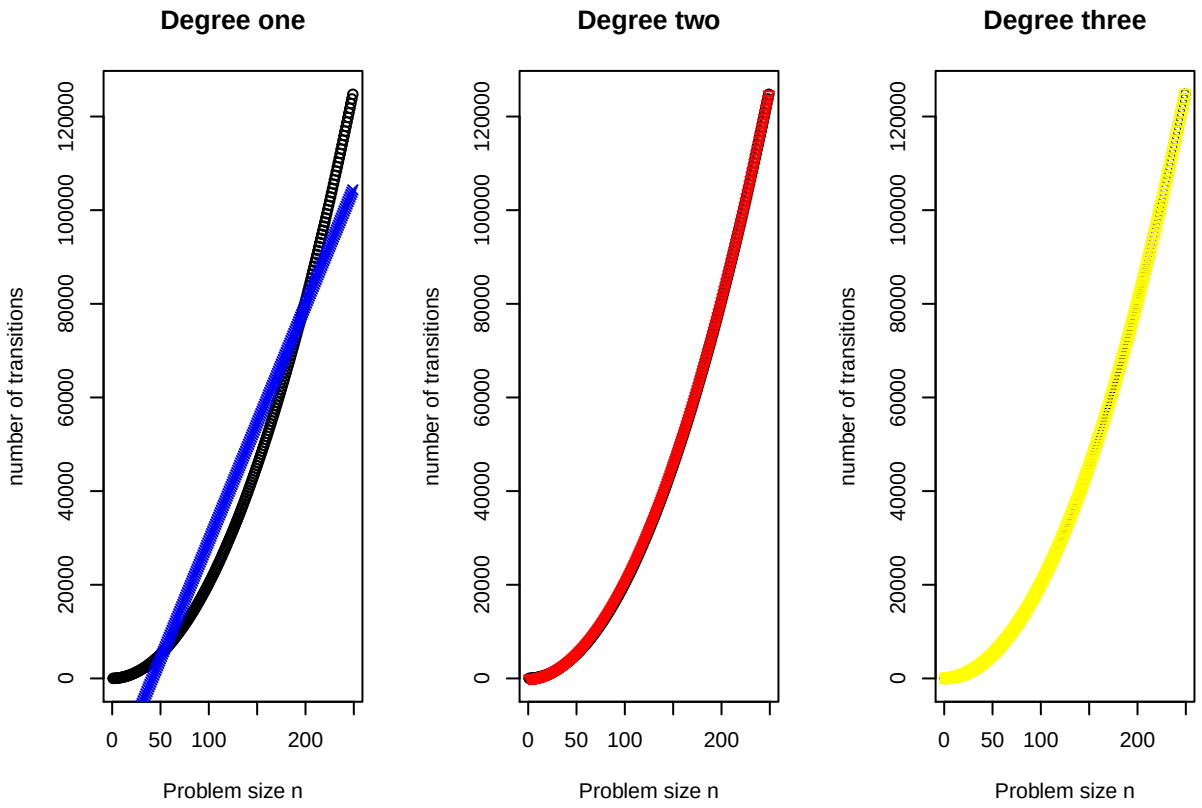


To analyse the data, we can fit different models. Assuming that there will be linear function combination of the problem sizes raised to some degree, several functions can be considered:

1. Function of first degree
2. Function of second degree
3. Function of third degree

Exponential functions are discarded as there is no exponential growth term.

```
# linear model degree 1
fit.one = lm(palindrome$transitions ~ poly(palindrome$problem_size,1))
# linear model degree 2
fit.two = lm(palindrome$transitions ~ poly(palindrome$problem_size,2))
# linear model degree 3
fit.three = lm(palindrome$transitions ~ poly(palindrome$problem_size,3))
par(mfrow=c(1,3))
plot_palindrome("Degree one")
points(palindrome$problem_size,
       fitted(fit.one),
       col="blue",pch=4)
plot_palindrome("Degree two")
points(palindrome$problem_size,
       fitted(fit.two),
       col="red",pch=6,cex=0.8)
plot_palindrome("Degree three")
points(palindrome$problem_size,
       fitted(fit.three),
       col="yellow",pch=0)
```



Clearly, degree two and degree three offer a good fit. The next step is to investigate the significance of the term raised to the power three. This can be done using an Analysis of Variance test.

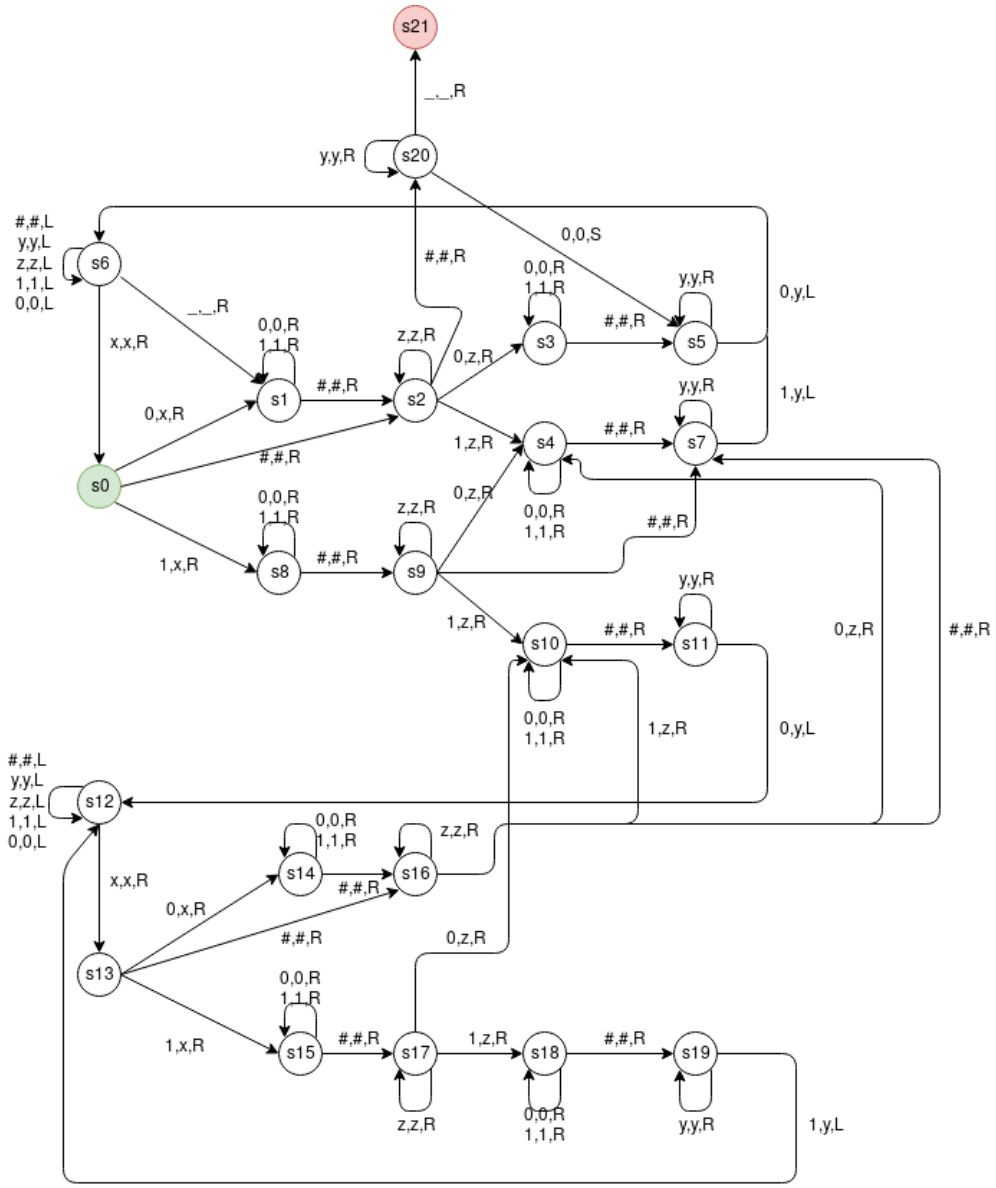
```
Analysis of Variance Table

Model 1: palindrome$transitions ~ poly(palindrome$problem_size, 2)
Model 2: palindrome$transitions ~ poly(palindrome$problem_size, 3)
  Res.Df        RSS Df  Sum of Sq      F  Pr(>F)
1    246 1.7349e-19
2    245 1.7016e-19  1 3.3332e-21 4.7992 0.02942 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Considering the output of the analysis and in particular the p value of 0.0294153 with a low significance for a model without any random factor, one can conclude that the significance of the term raised to the third degree does offer a better fit, but the cost of the extra precision outweigh the advantages. Hence the quadratic model is the preferred model.

The conclusion of this analysis is that the experimental analysis of the palindrome algorithm is bounded by a quadratic function.

Binary addition

This TM tests for inputs of the form x#y#z, where x,y,z are binary numbers with the least significant bit first. The TM accepts the input if x+y=z. This particular implementation also accepts if either x or y are left out, e.g. x##z is a valid input, given x=z. When implementing this description, care had to be taken when dealing with carries. Drawing the diagram yields a complicated structure, which is difficult to grasp.



The state marked in red describes the accepting state, green is the start state. One can spot that the graph is roughly split into two very similar areas, this is because a carry is assumed in the lower half.

Given an TM = (Q,Σ,I,q,δ,F), here we have:

Q = {s0,s1,. . . ,s21}
Σ = {I,_,x,y,z}
I = {0,1,#}
q = {s0}
δ = as defined in the diagram
F = {s21}

The file describing the very same version encoded in a machine readable format is named “addition.txt” and included in this submission.

The input size is defined to be the length of one of the binary numbers. Given size n of one of the binary numbers, the total input string will have a length of $3 * n + 2$. The 2 accounts for the #’s.

Validity of implementation

The validity of this implementation has been tested with a java unittest suite. Beyond the examples given in the specification, this test suite includes more sophisticated corner cases, including additions with a double carry, with one of the numbers left empty or symbols not included in the alphabet. Further, this example has been tested by automatically generated inputs. The automatic generation of a addition is fairly tricky, so some extra care had to be taken. At first two random integers are created, they can either be one or zero. These are appended to a string. This step is repeated until the string has the required size. Then these strings are turned into BigInteger objects and are added to retrieve the result. However, the result can be shorter, given that both most significant bits can be zero and hence the result of the addition will be zero. This is not handled by the TM, so an extra 1 needs to be attached. Finally, the strings will be reversed, to have the least significant bit on the left most side and the strings are concatenated into the right format with a # as a separator.

Theoretical complexity analysis

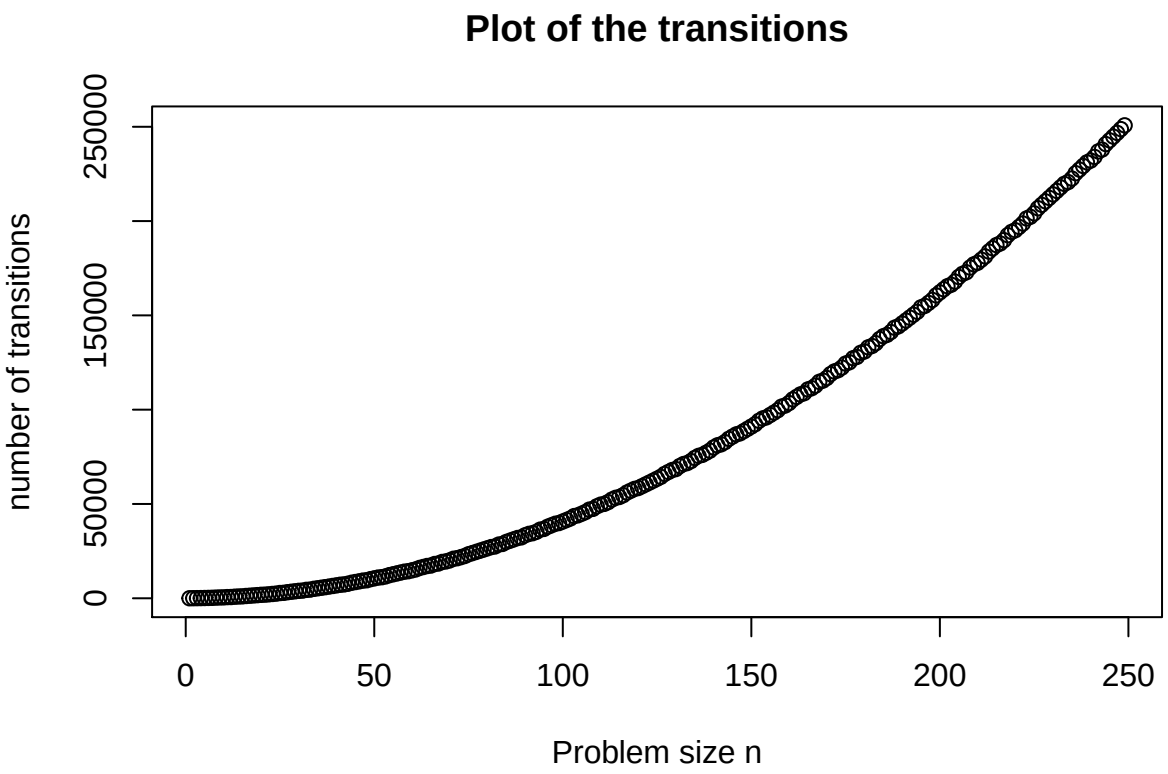
Even thought the diagram displays a complicated structure, the underlying moves of the head on the tape are not as complicated to understand. The diagram has such a complex structure because one needs to persist the first value in a state in the first steps and set up an equivalent structure for the case in which there is a carry. Simplifying the structure and yields an algorithm which goes back and forth between the end points and with each iteration the head needs to traverse one symbol less, as the outermost symbols are replaced by the empty symbol. As described a problem of size n will yield an input string of size $3 * n + 2$. As per usual with complexity analysis, one can drop the constant and one can also drop the factor 3. Hence:

$$N = (n) + (n-1) + ... + 1 = \sum_{i=1}^{n} (n-i) \leq n + n + ... + n$$

Similarly to the initial problem we obtain a theoretical complexity, which is upper bounded by $O(n^2)$.

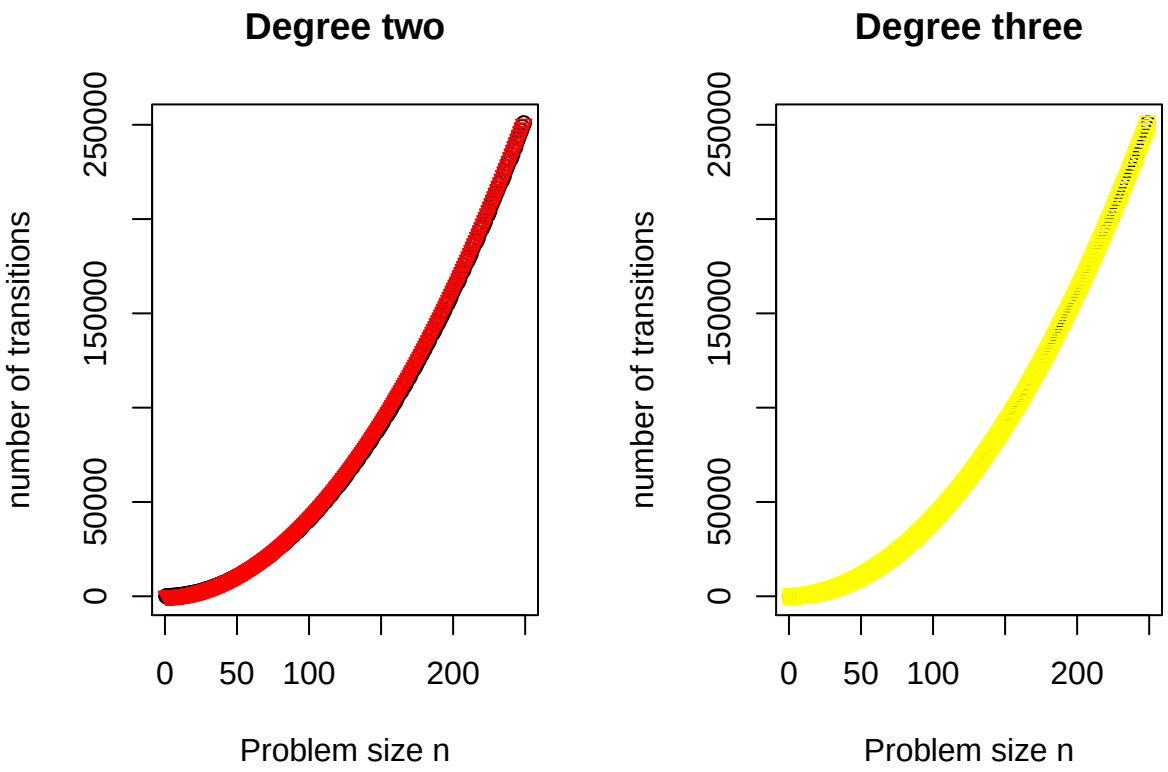
Experimental complexity analysis

As defined above, input size of 1, will actually be a string of length $3 * n + 2 == 3 * 1 + 2 = 5$. E.g. the string "1#0#1" is input size one, as it is one loop in the algorithm. This is a difference which is not relevant in the theoretical analysis becomes relevant in practice when predicting the accurate count of transitions. The sampled data is displayed below:



As the plot looks similar in shape the one observed before when investigating the complexity of the palindrome algorithm, the seems to be no necessity to investigate the linear model any further.

```
# linear model degree 2
fit.two = lm(transitions ~ poly(problem_size,2),data=addition)
# linear model degree 3
fit.three = lm(transitions ~ poly(problem_size,3),data=addition)
par(mfrow=c(1,2))
plot_addition("Degree two")
points(addition$problem_size,
       fitted(fit.two),
       col="red",pch=6,cex=0.8)
plot_addition("Degree three")
points(addition$problem_size,
       fitted(fit.three),
       col="yellow",pch=0)
```



Very similar to the plots above, both models seem to offer a good fit. However, this time, there is more significance to the cubic factor which can be seen in the ANOVA below:

```
Analysis of Variance Table

Model 1: transitions ~ poly(problem_size, 2)
Model 2: transitions ~ poly(problem_size, 3)
  Res.Df      RSS Df Sum of Sq      F   Pr(>F)
1    246 20378587
2    245 19549084  1    829503 10.396 0.001435 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

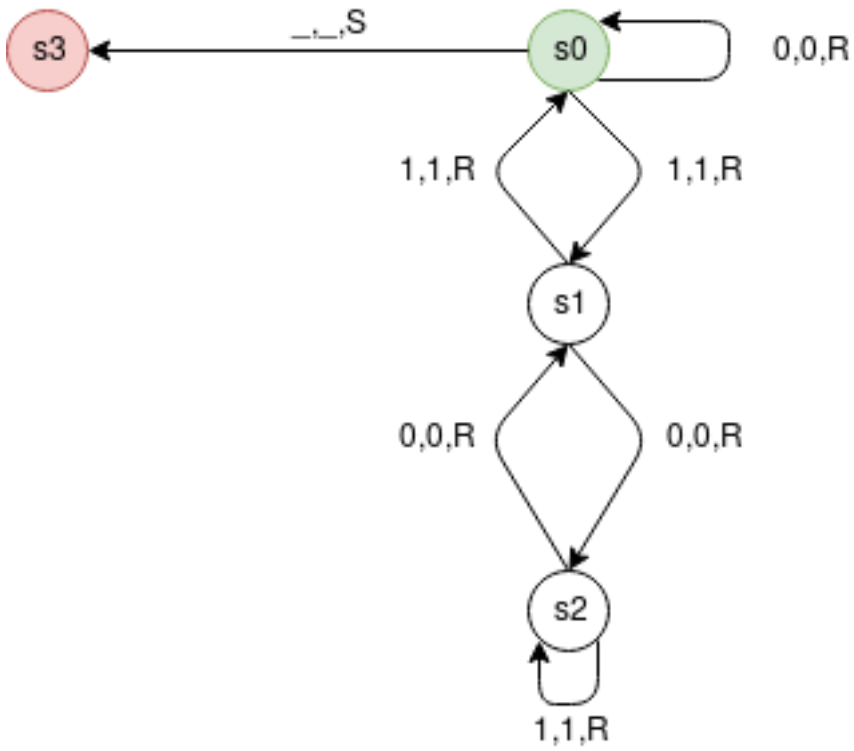
The p value for the cubic term in this is now 0.0014347. This is a significance level at which it is down to the reader or researcher if this extra level should be included in the model. There are strong arguments for the inclusion of the term raised to the power three, including a more accurate prediction. Predicting the last observed transitions with the fitted models and taking the differences between the observed and the predicted model yields the following:

Size	Observed	Cubic Model	Error	Quadratic Model	Error
244	240838	240602.1	235.9457	240487.6	350.3994
245	242805	242576.6	228.4173	242455.5	349.5072
246	244780	244559.3	220.7243	244431.4	348.5929
247	246763	246550.1	212.8655	246415.3	347.6563
248	248754	248549.2	204.8397	248407.3	346.6976
249	250753	250556.4	196.6457	250407.3	345.7167

Clearly the cubic offers a better fit. However, considering the extra complexity in calculating the model and comparing the differences to the observed numbers, the additional value seems small.

Divisibility by 3

This TM tests for inputs of the form x, where x is a binary number with the least significant bit first. The TM accepts the given string if the string is divisible by 3 and rejects if the string is not divisible by 3. A transition graph is drawn below:



The state marked in red describes the accepting state, green is the start state.

Given an TM = (Q,Σ,I,q,δ,F), here we have:

Q = {s0,s1,s2,s3}
Σ = {I,_}
I = {0,1}
q = {s0}
δ = as defined in the diagram
F = {s3}

The file describing the very same version encoded in a machine readable format is named “divide.txt” and included in this submission.

The input size is defined to be the length of the string to check.

Validity of the TM

The implementation of this TM has been tested with a unittest suite. The test suite is considerably smaller than the other ones, as the small amount of states, requires less coverage. The implementation passes all tests. Further, there have been samples taken from the automatised number generation and a counter check has been done in R if there randomly generated numbers are in fact divisible by 3.

Theoretical complexity analysis

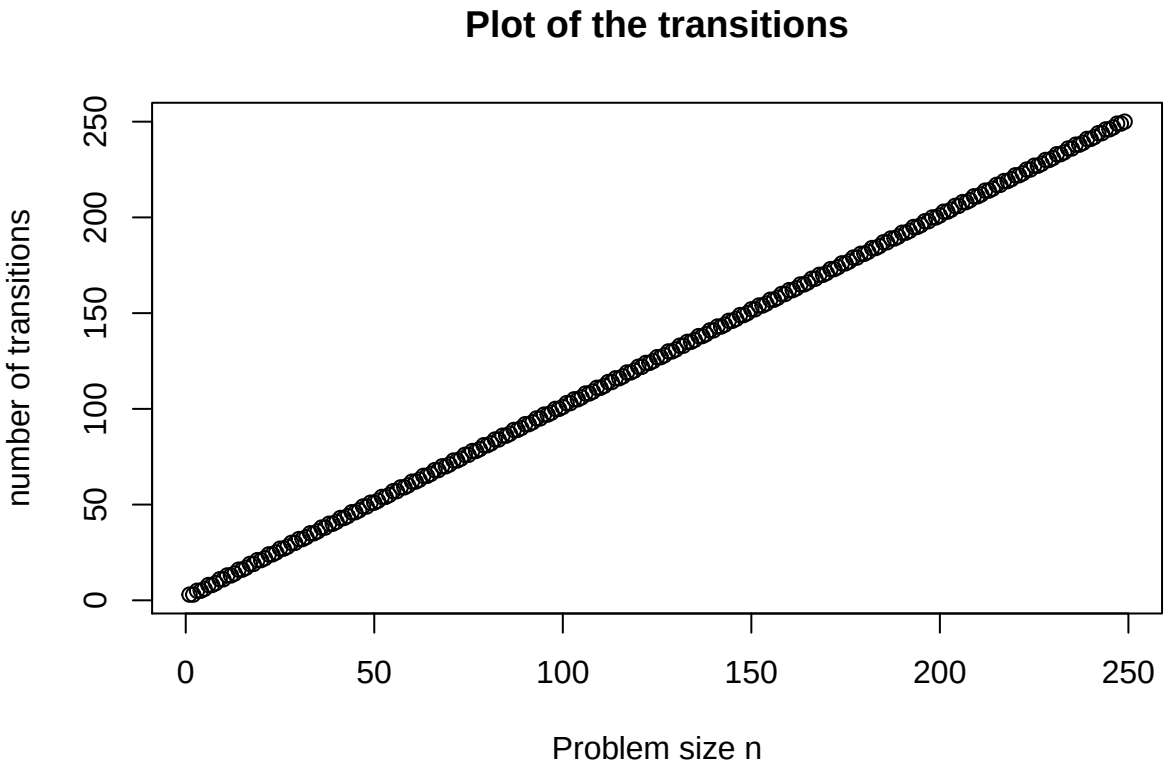
Given a number of size n, the algorithm will iterate through the whole input string. When the head reaches the end of the input string there are two options:

a. the machine is in s3
b. the machine is not

In case a, the string is accepted, in case b, the string is rejected. Hence the string only needs to be traversed once, hence there will be exactly n transitions plus the final one into s3. This translates into $n + 1 = O(n)$.

Experimental complexity analysis

As seen in the theoretical analysis, division by three has a function, which is linear in theory without dropping constants. This straight line is also retrieved when experimenting with divisibility by 3 in practice. The plot below plots transitions against input sizes.

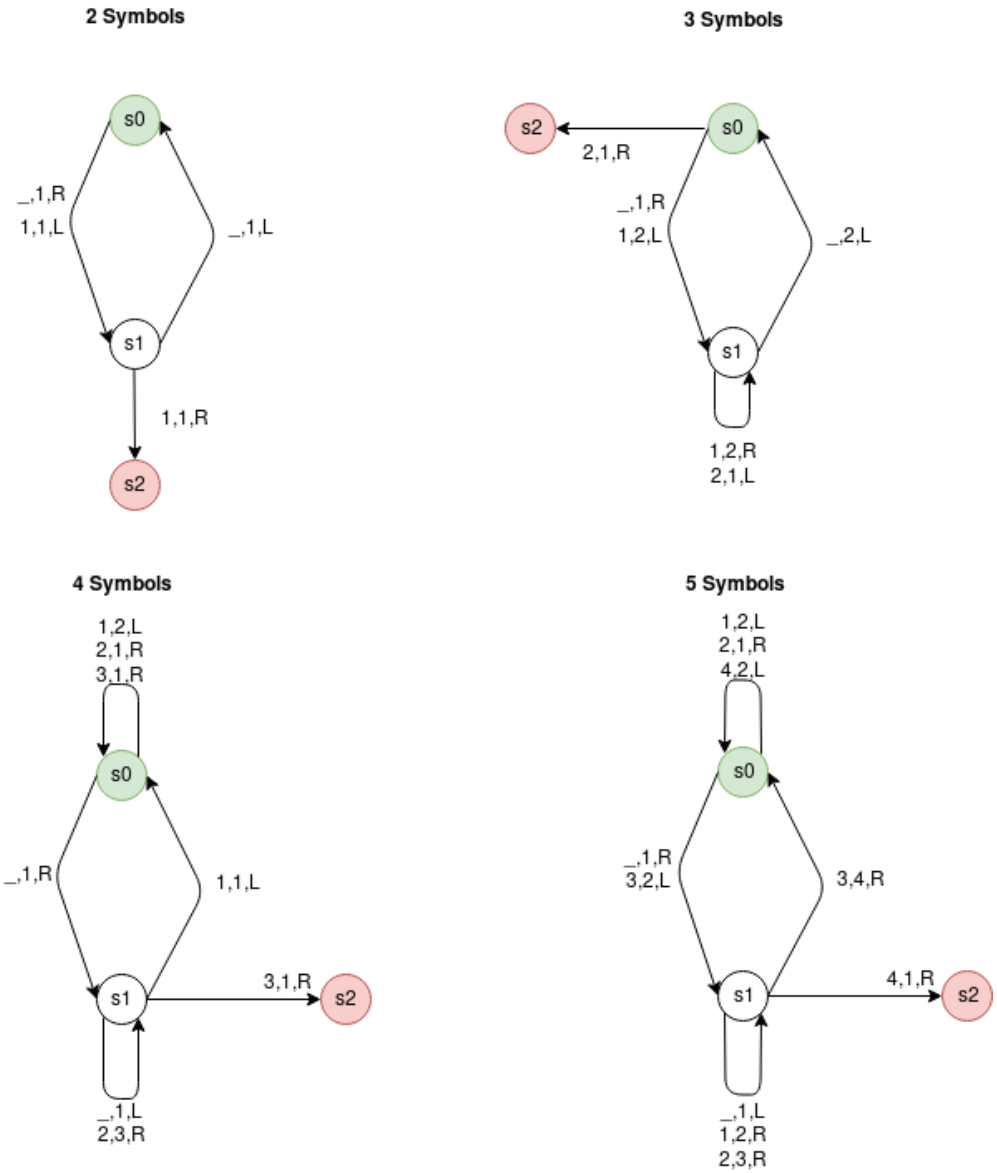


Given the straight line and the theoretical analysis, there is no further investigation worth the time and it can be concluded that the problem divisibility by three has a linear complexity checking time.

Busy beaver

The Busy Beaver is a Turing Machine of a specified input size, which takes as many transitions as possible without entering an infinite loop. As this varies depending on number of states and the size of the input alphabet, there is one TM for each combination of the above. Known to increase very rapidly, there are four Turing Machines implemented in this analysis. They all have two states, but the size of the alphabet increases from two up to and including five. Another crucial difference to the other problems above is that the busy beaver TM's always start with an empty tape.

The TM's are all slightly different in their transitions, but have the common characteristic of two states.



The design of the different machines were taken from other resources, in particular:

1. The 2 state 2 symbol machine was designed by Rado [1]
2. The 3 symbol machine was taken from Lafitte [2]
3. The 4 symbol machine was taken from Ligocki [3]
4. The 5 symbol machine was taken from Ligocki [3]

From now on the different TM will be referred to as (state,size of input alphabet)BB, where BB stands for busy beaver. The 2 state 2 symbol TM is a (2,2)BB, as an example.

Theoretical complexity analysis

The theoretical analysis of the Turing Machines themselves on their own is beyond the scope of this submission, however, there are a few things worth highlighting. Firstly, it is fairly easy to grasp the basic concept of the initial (2,2)BB. The TM builds up a row of three ones before finishing. The empty tape develops in the following way, where x denotes the empty _:

Case	Tape	State
1	xx**x**xx	0
2	xx1**x**x	1
3	xx**1**1x	0
4	x**x**11x	1
5	x1**1**1x	0
6	x**1**11x	1
7	x1**1**1x	2 - Halted

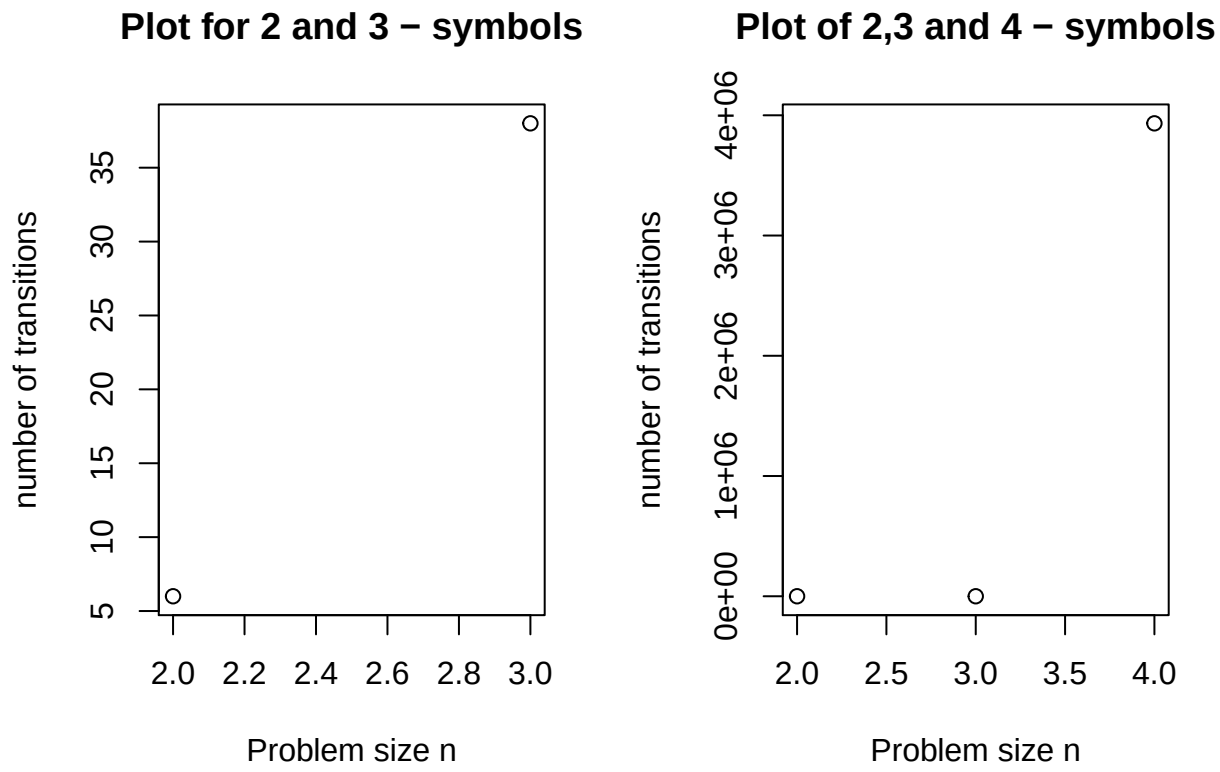
The main idea of this TM is to create a string on the tape through which it then has to iterate again. There are 7 cases denoted in the table above, which let us conclude that there will be 6 transitions required to write the 111 string on the tape. According to research, this is as many transitions as possible for a TM with these specifications. The idea seems simple for a (2,2)BB, but as soon as the input alphabet is increased by one sign and there is a (2,3)BB the situation becomes more complicated. Lafitte published in his 2007 paper, that the maximum amount of transitions are 38 and one will end up with an input string of length 9 on the tape. This is a steep increase for an additional input letter. The idea underlying the (2,3)BB is that the input string of a certain length is written and then the ones and twos will be replaced with each other while iterating through the string. The same idea applies for the next increase in complexity. The (2,4)BB has already 3932964 transitions and then the (2,5)BB has $1.9 * 10^{704}$ transitions as published by Ligocki[3]. The count of transitions for the (2,5)BB is so large that the number exceeds the atoms in the observable universe by a factor of more than 10^{600} [4]. A number too big to even grasp. According to Ligocki a (2,6)BB reaches more than $2.4 * 10^{9866}$ transitions. Both numbers will be put into perspective in the next section.

Experimental complexity analysis

The theoretical analysis given above can be found one to one in the results of the experiments. This is a table of the results:

problem_size	transitions
2	6
3	38
4	3932964

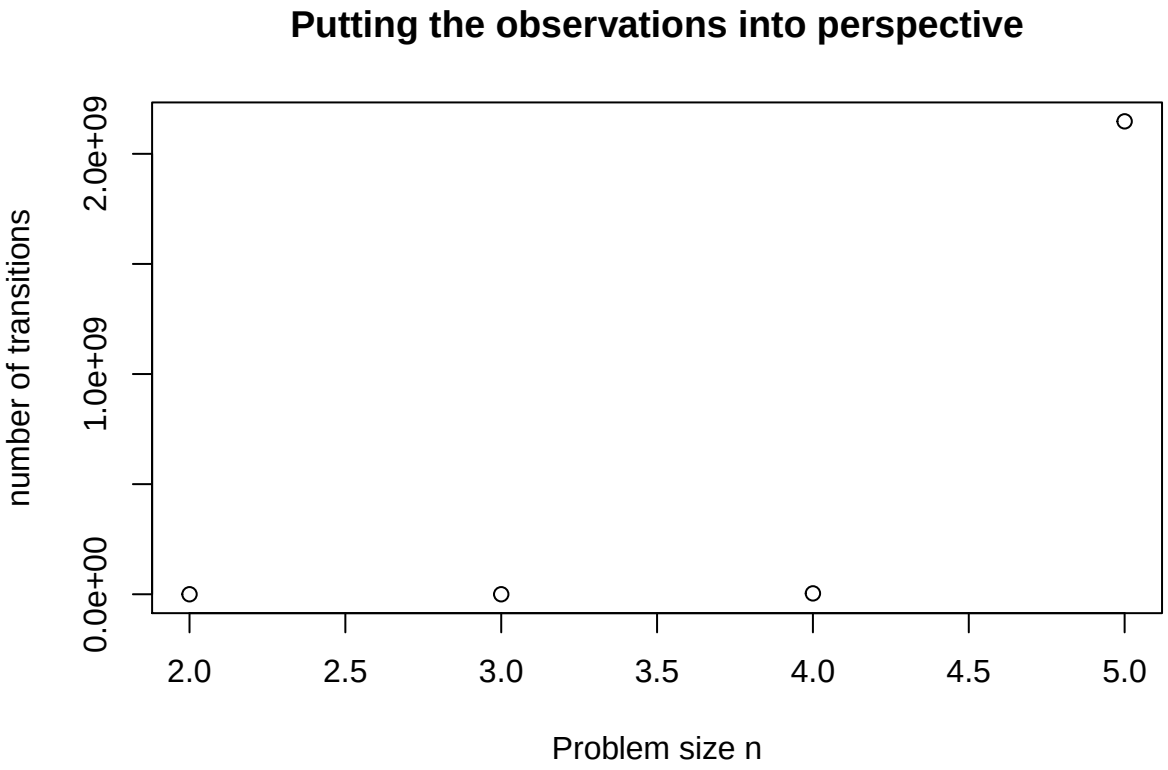
The numbers are exactly as expected from the theoretical analysis. Setting these numbers in perspective to each other gives an impressive plot:



The growth of the number of transitions is difficult to grasp by just looking at the increase of the symbol count from 2 to 3, but setting it into perspective with the transitions needed for input size 4, gives of a completely different picture. Please note the right and the left diagrams above.

Interestingly, taking the time of the TM with 4 symbols took 2463571311 nanoseconds which are 0.04 minutes. Assuming each transition takes the same amount of time, we can now calculate how much time it would take to theoretically calculate the time for the TM with 5 symbols. This now explains why the theoretical analysis is only a theory but not available in practice. The factor the transitions increase by is roughly $10^7 98$. However, onlt taking 10^7 as a factor would lead to a computation almost taking a year. Computing the (2,5)BB would take hundreds of years on the lab machines and even a faster machine would not improve the computational execution to a bearable level.

For an exponential growth this function would need to have the same factors between each increase in step size, which is not the case! This means that the observed busy beaver functions grow faster than an exponential function. Putting the observed data into perspective with the maximum value R can compute, which is still much smaller than the theoretical values for (2,5)BB, let alone (2,6)BB, gives the following graphs:



This makes the busy beaver TMs interesting and unique. The complexity is so rapidly increasing that a (2,8)BB has yet to be found. The busy beaver function is also regarded as uncomputable for the reasons above.

Comparison of the Problems

Before turning to the summary of this chapter, there are a few interesting facts to highlight. Notably that the difference in transition count for the Palindrome problem and the Addition is quite noticeable even though both are bounded by $O(n^2)$. This is down to the factors involved when conducting the experiments and the factors dropped when defining input size. Especially dropping the factor multiplication factor from the definition for the length of the input string of the addition problem causes a strong level of skewness in when comparing the results. To compare the problems the problem size needs ot be defined more stringent.

Summary

The four problems present are four very different algorithms with different complexities presenting the power and variablilty of Turing Machines. The given problems in the specifications display how the complexity of seemingly easy tasks become more difficult when just one tape is available and the problem size increases. On the first look checking an addition for a correct solution seems a lot more difficult than a palindrome check, but as seen above both are bounded by the same $O(.)$. Finally, the divisibility check shows how even simple TMs can produce helpful tasks and the busy beavers provide a taster into the theoretical field of general computational bounds.

Non deterministic TM

Another interesting thought is to simulate a non-deterministic Turing Machine NTM with a deterministic Turing Machine. One of the most striking differences and most defining factor of a NTM is the "guessing" ability of the NTM. The NTM can be in multiple states at the same time and will accept the input when one of the different routes will lead to an accepting state. In an idealised world all of these different paths would be executed in parrallel, allowing exponential algorithms to be run in polynomial time, which would be a breakthrough for many computational intensive processes.

There are multiple ways to encode this into a non deterministic machine:

1. Multiple configuration states: Creating a list of states the Turing Machine is in.
2. A three tape approach in which each tape serves its purpose in computing the input string and saving the state.

For the purpose of this submission, a forked version of the first of the two approaches has been implemented. Recursively the implementation will call a "run" initiator on itself and will run the next steps for every single possible transition given the current state. Running time for simulators like this might be higher, but the count of transitions will be lower, which is the important note on this problem.

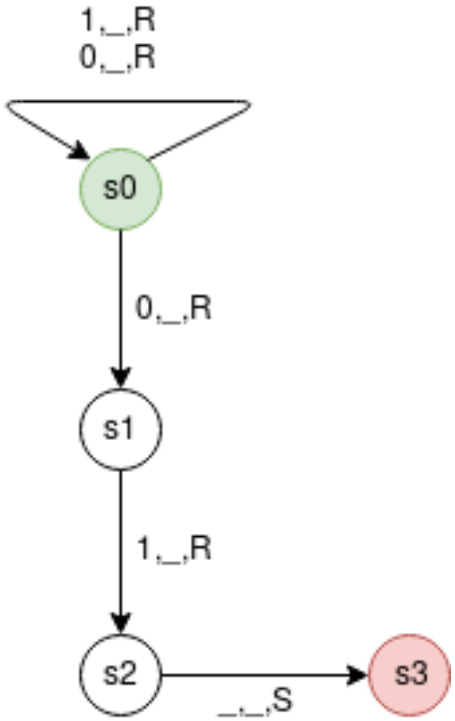
Design

The design is very similar to the design of the TM in the first tasks, however, this time the function to execute the next steps is called recursively on the same object. However, the current state of the tape, the current head position and the current state do not count for the whole TM anymore, but are only valid for the lifetime of the variable within the function. The design still uses HashSets to keep all states and transitions in order. This allows the progress to bounce back and proceed from the latest valid state to try to the next possible transition from that state. It will fail if the possibilities of transitions run out, if from the original call no paths succeed, than the whole input will be rejected.

Binary word

The problem to recognize a string with an unlimited amount of 0's and 1's before the string ends in 01 has been chose. This problem already occurred in lectures and allows to demonstrate the advantage of a NTM over a TM. Note that this could also be encoded in an NFA. The description is included in the file "ntm_language.txt".

The NTM has the following transition graph:



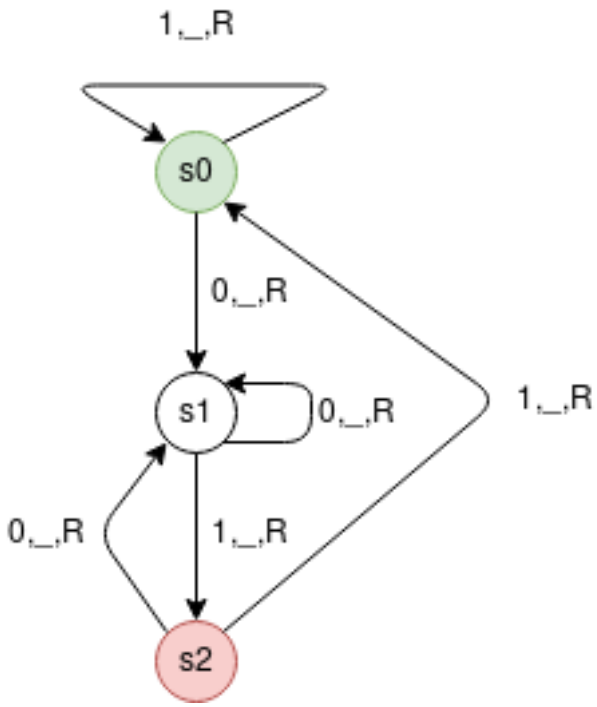
Further it has the following formal definition NTM = (Q,Σ,I,q,δ,F):

Q = {s0,s1,s2}
Σ = {I,_}
I = {0,1}
q = {s0}
δ = as defined in the diagram
F = {s2}

Please note that what makes this problem non deterministic is the fact that in s0 for the input 0, the machine can either stay in s0 or change into s1.

Theoretical Analysis

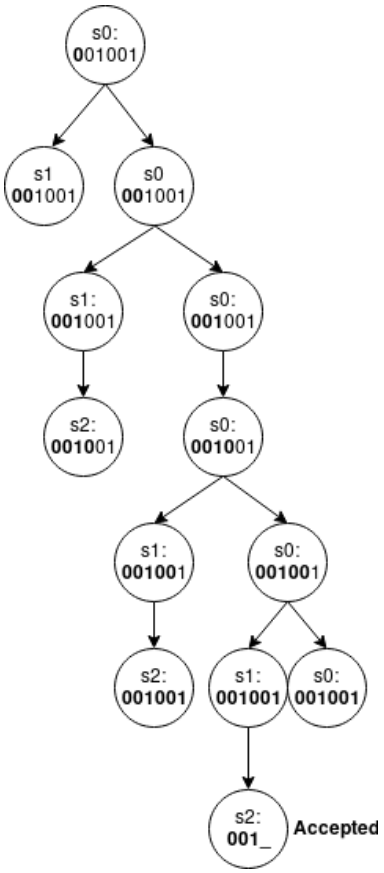
In non deterministic Turing machines the underlying assumption is that the Turing Machine always makes the perfect guess. Hence, we can discard all the wrong guesses in this case. For an input of size of n, the NTM will iterate through all the different digits in the binary string. After the last integer the NTM will either accept the string or fail. This will give the NTM exactly n transitions. Hence, this complexity is bounded by $O(n)$. This NTM could also be easily implemented as a deterministic Turing Machine. This has a theoretical justification as shown by Sipser [5]. The most interesting difference between the two implementations of this problem, which could be implemented as deterministic finite state automata, is the count of total possible transitions between states involved. The deterministic version of the algorithm is displayed below:



The differences are striking: Fewer transitions have to be specified in the NTM while the outcome will be the same. As per usual the simpler model is the favoured model if the same outcome is achieved and this is were the power of a NTM comes in even for deterministic TM solutions.

Experimental Analysis

Computing this non - deterministic problem can in fact yield multiple paths. The best way to display the path is a tree. Given the string “001001” we can compute the following graph with the different endpoints.



The encoded NTM will also iterate through the same tree. We see in this case that the input length is 6 and that the shortest path in tree is also 6 transitions long. The encoded NTM will also offer a similar insight into the processes and a screenshot has been attached in the appendix. The relation $inputsize \approx transitions$ was not subject to change. Hence, this problems grows linearly with input size.

Conclusion

This report has shown how versatile Turing Machines are. In particular the set of problems which can be encoded into a one tape Turing Machine problem and the different performances of those algorithms. The limits of these complexities are set by the busy beaver functions growing faster than any other computable function. Another important conclusion can be drawn from the results, which is that non-deterministic Turing Machines do not extend the field of computable problems, but offer a theoretical shortcut to many problems.

Bibliography

[1] Rado T. (1963) On a simple source for non-computable functions in: Proceedings of the Symposium on Mathematical Theory of Automata, April 1962, Polytechnic Institute of Brooklyn, April 1963, 75-81.
[2] Lafitte G. and Papazian C. (2007) The fabric of small Turing machines in: Computation and Logic in the Real World, Proceedings of the Third Conference on Computability in Europe, June 2007, 219-227.
[3] Michel P. (2017) The Busy Beaver Competition: a historical survey
[4] Universe today, How many atoms are there in the universe? (2009) https://www.universetoday.com/36302/atoms-in-the-universe/
[5] Sipser, Introduction to the Theory of Computation

Appendix

Output for a NTM; the NTM bounces back to the last valid state when no valid transition is found