CS3052

150001088

25 April 2018

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 executed on the problem of Palindrome regognition, Binary addition, BLANK and BLANK.

Reproducability

All my experiments are reproducible with the Code submitted. All relevant data files are created when the main method is run. BLANK [content about running the machine] This report has been created with R markdown and the plots in this project have been made with different R libraries.

Design

This section highlights the design decisions made to encode a Turing Machine in Java and then how the experiemts are conducted.

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 trasitions 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 ckeck 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 were specified before. This has been discussed with the marker before submission.

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 containting the description of the turing machine and the output file name for the results. The name TuringTimer can be considered slightly missleading 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 descriped in the previous chapter and increases the problem size step by step. Problem size should not be confused with the input lenght 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.

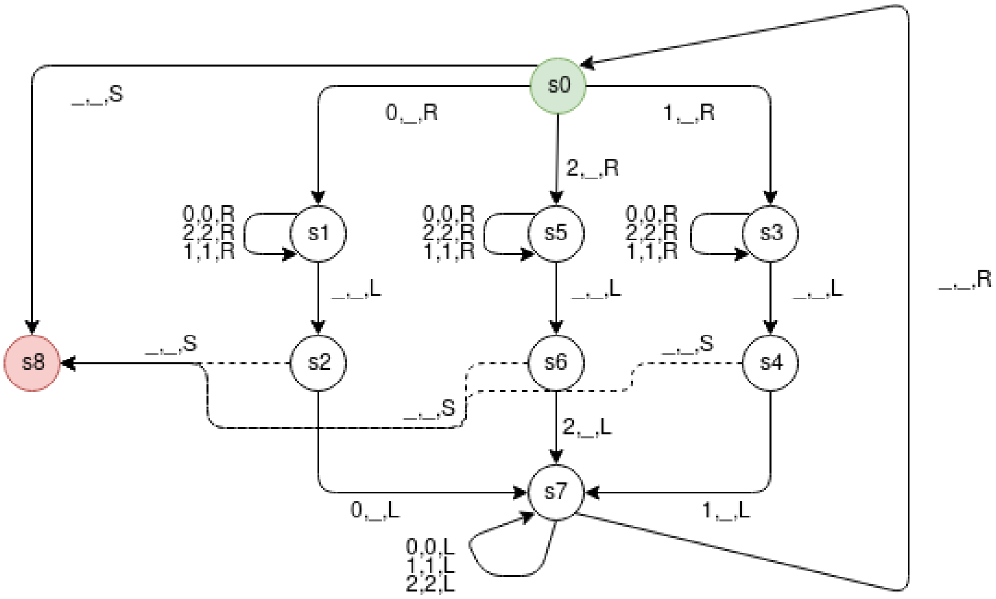
The creation of problems is randomised with seed 1234 and fitted to the input file name. The file descriptor is analysed and the problem of the specified size is returned, all returned problems will be accepted ultimately. There is no cleaning of rejected inputs. The output file contains the input which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

The Problems

In the subsequent sections 4 problems will be analysed. For each of the problems a unittest class is implemented to prove the validity of the developed TM.

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 restrived 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.

Theoretical complexity analysis

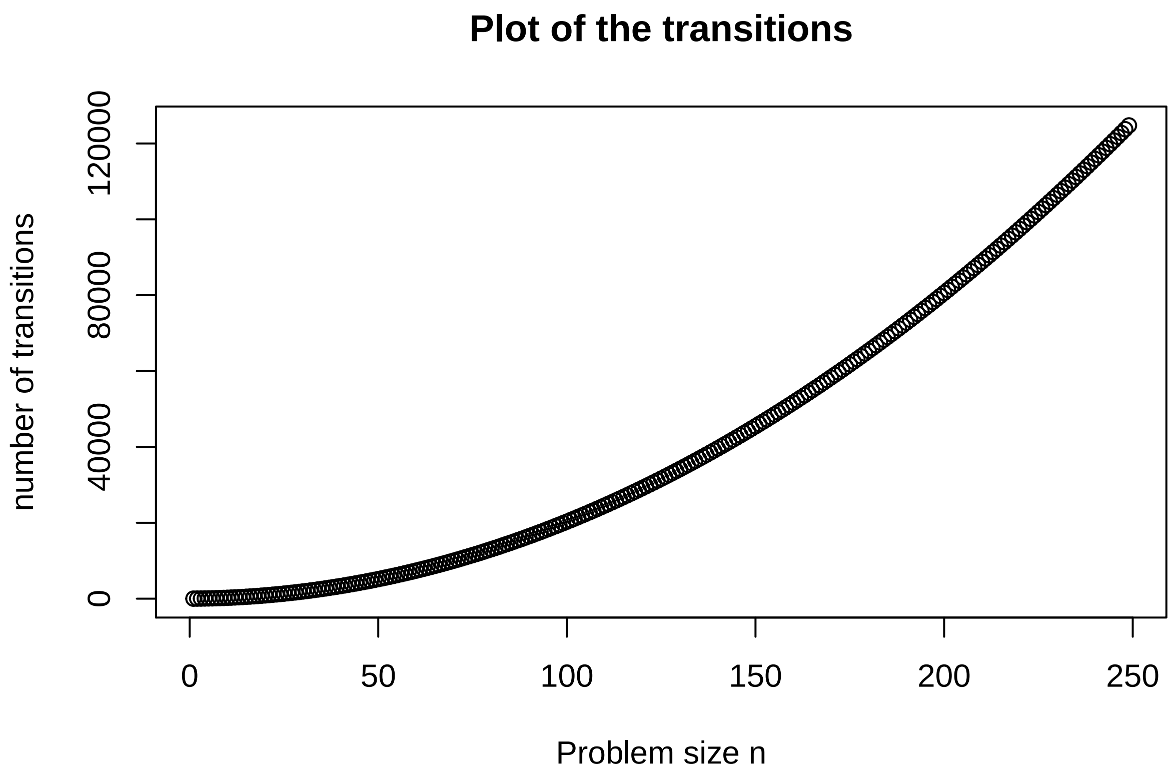
With every single correct input, there will always be at least one transition. This is also true for the empty input. This is because the state needs to change at least ones into the accepting state. For an input size of n, the algorithm will read the first input and then traverse until the head points to an empty symbol and then the head will travers 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 inquality $N \leq n^2$ 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, as it requires one loop in the algorithm. The sampled data is displayed below:

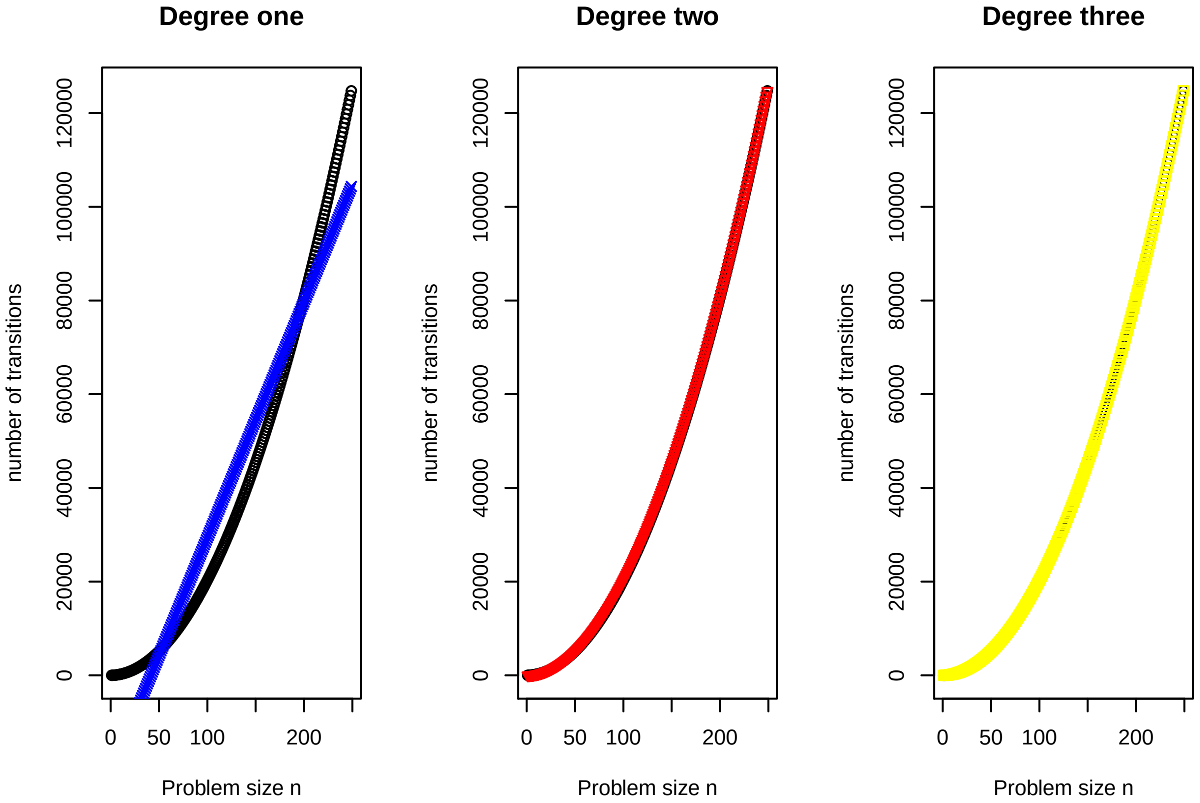


To analyse the data, we can fit different models. Assuming that there will be linear function combination of the problem sizes raised to some degree, several functions can be considered:

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.

```r
# 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.

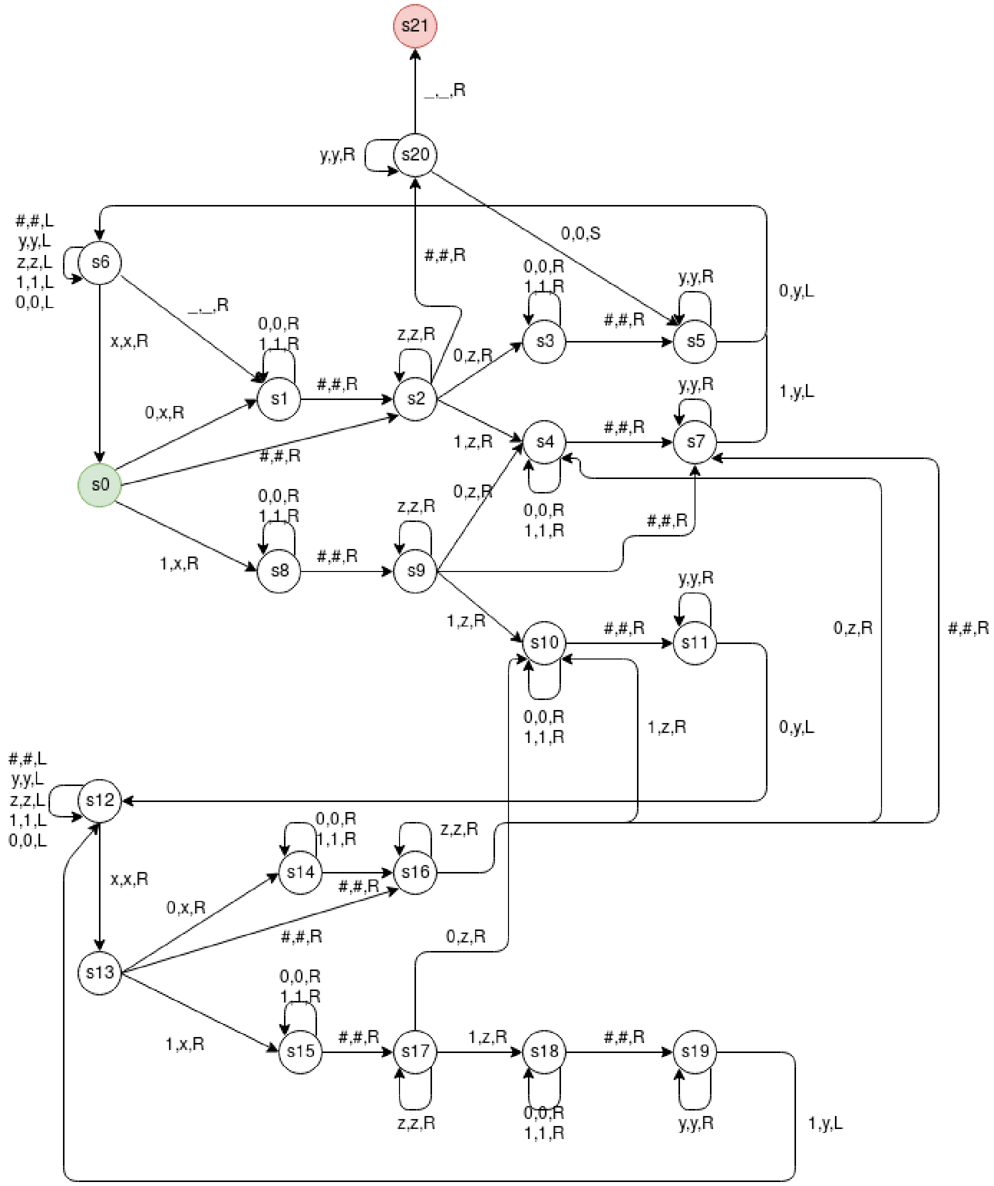
```
Analysis of Variance Table

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 outweight 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.

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

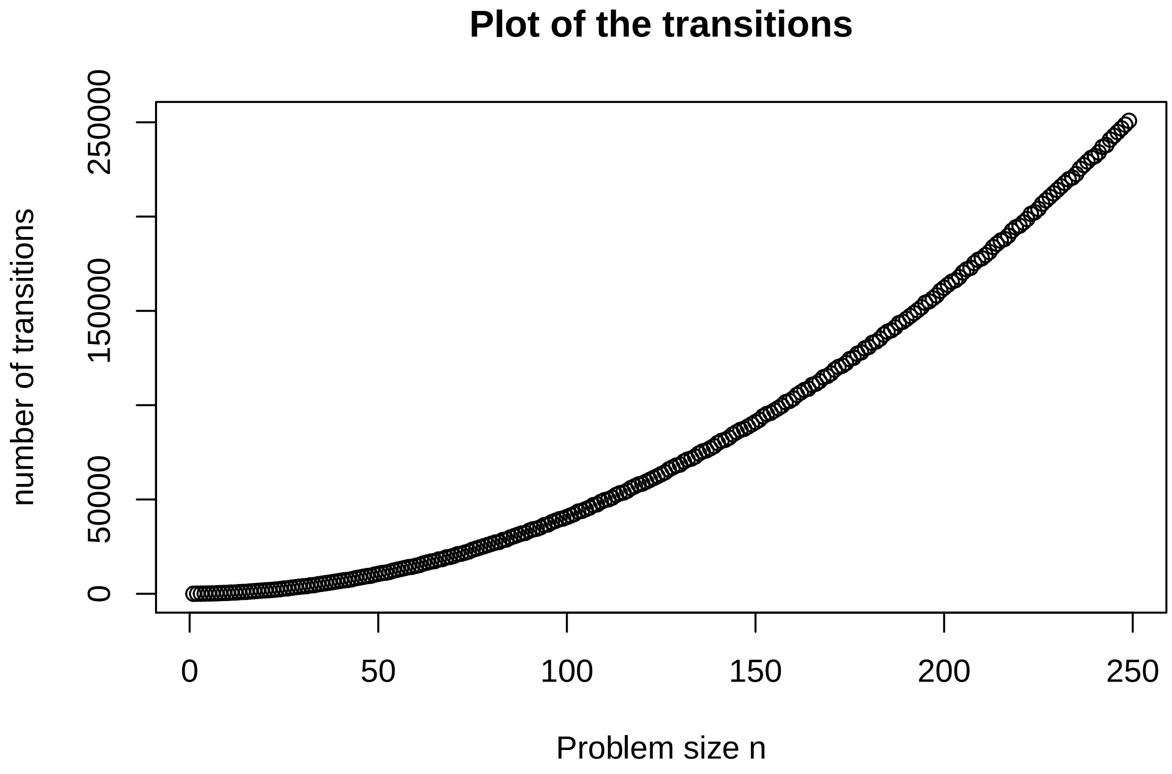
Even thought the diagram displays a complicated structure, the underlying moves of the head on the tape are not as complicated to understand. The diagram has such a complex structure because one needs to persist the first value in a state in the first steps and set up an equivalent structure for the 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 ususal 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 inital problem we obtain a theoretical complexity, which is upper bounded by $O(n^2)$.

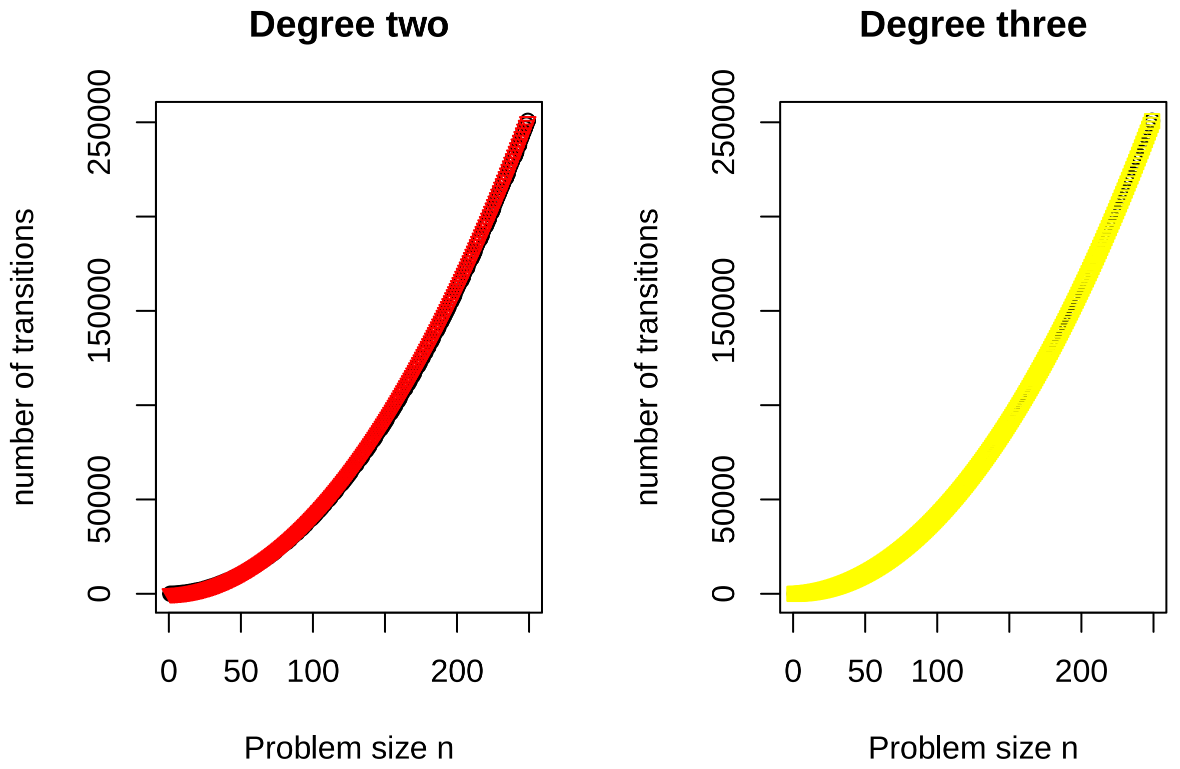
Experimental complexity analysis

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 pratice 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 nessecity to investigate the linear model any further.

```r
# 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
```

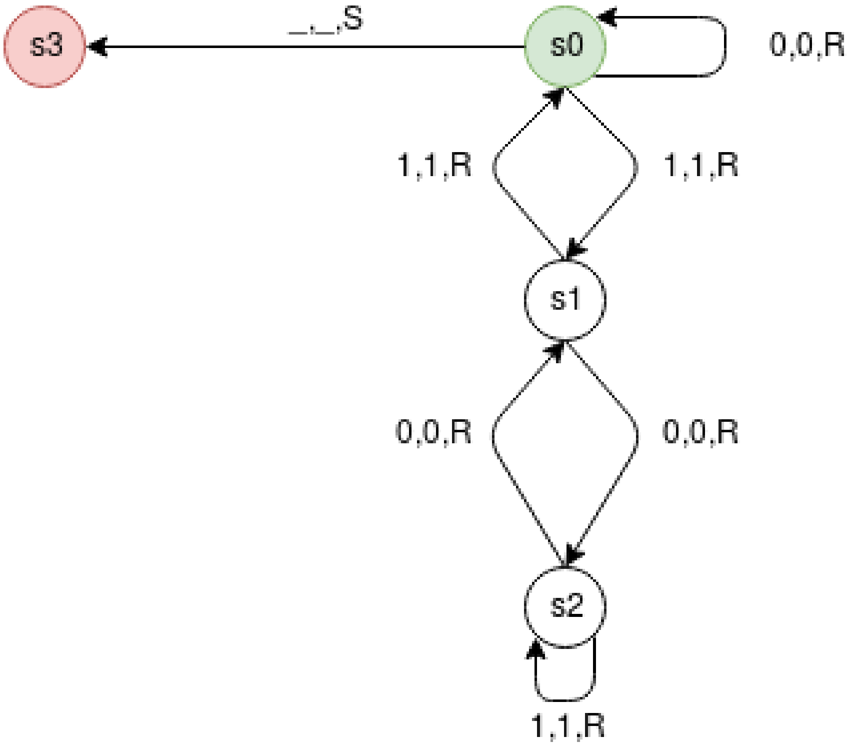
The p value for the cubic term in this is now 0.0014347. This is a significance level at which it is down to the reader or researcher if this extra level should be included in the model. There are strong arguments for the inclusion of the term raised to the power three, including a more accurate prediction. 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 oberserved 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.

Validility 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 implemenation 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

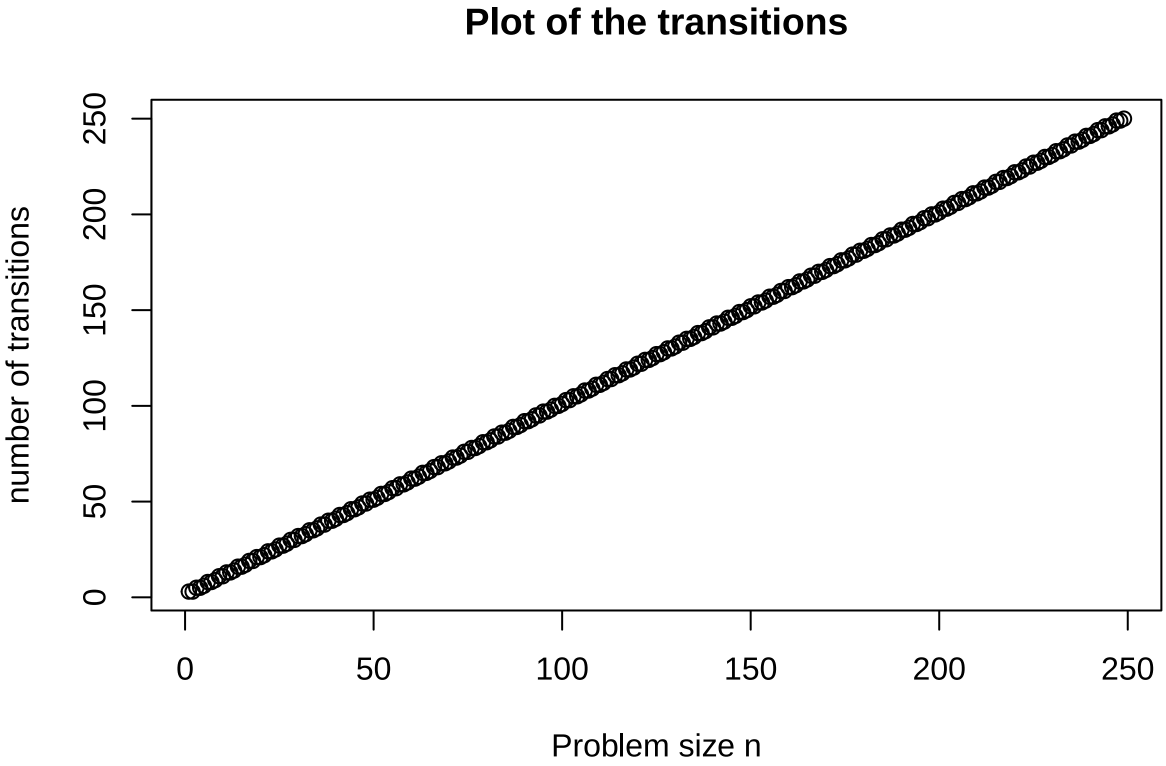
Given a number of size n, the algorithm will iterate through the whole input string. When the head reaches the end of the input string there are two options:

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 divisibilty by 3 in pratice. The plot below plots transitiions agains input sizes.



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

Comparison of the Problems

cubic more relevant for addition analysis –> bigger numbers, bigger discrepacy between theoretical (sacking the numbers/ constants) and the actual representation.

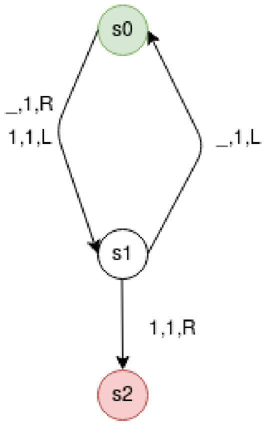
Busy beaver

http://turingmachine.io/ http://www.logique.jussieu.fr/~michel/ha.html

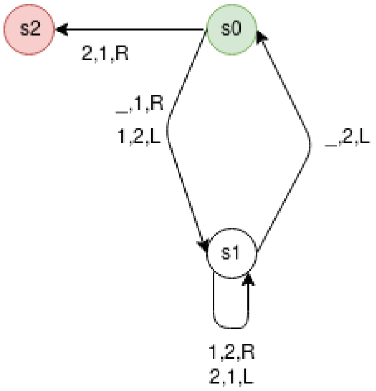
The Busy Beaver is a Turing Machine of a specified input size, which takes as many transitions as possible without entering an infinite loop. As this varys depending on number of states adn the size of the input alphabet, there is one TM for each combination of the above. Known to increase very rapidely, 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.

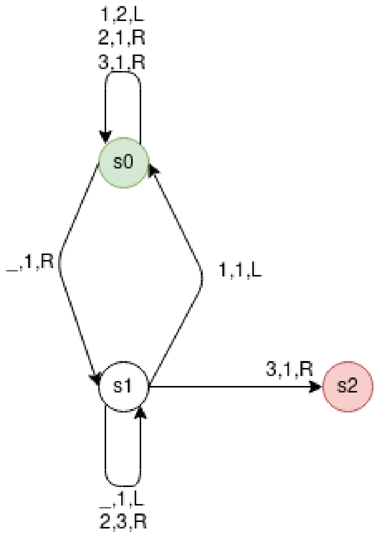
2 Symbols



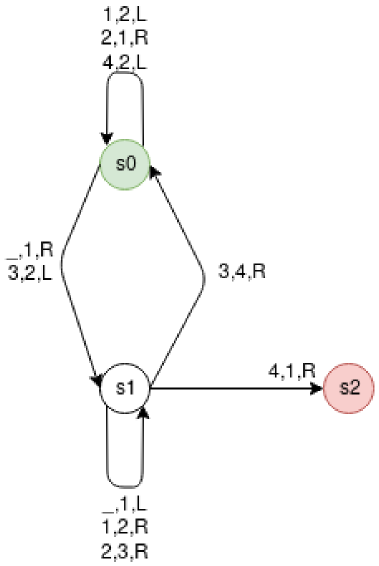
3 Symbols



4 Symbols



5 Symbols



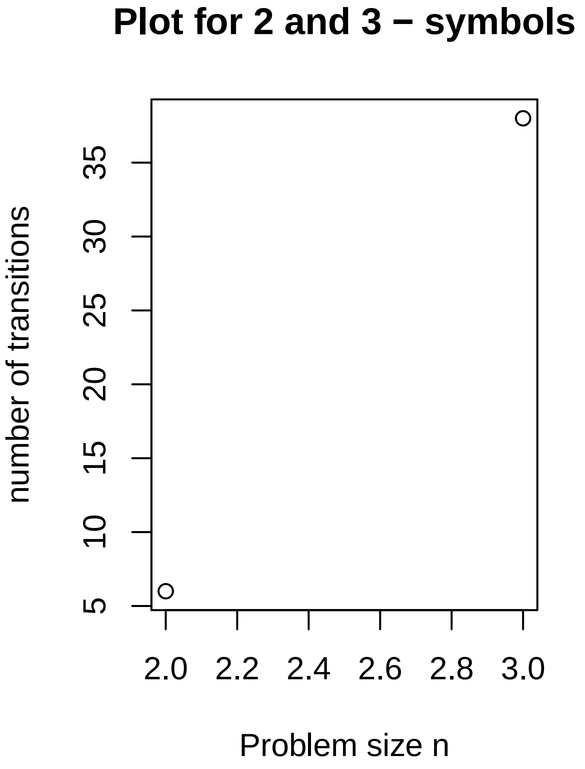
Theoretical complexity analysis

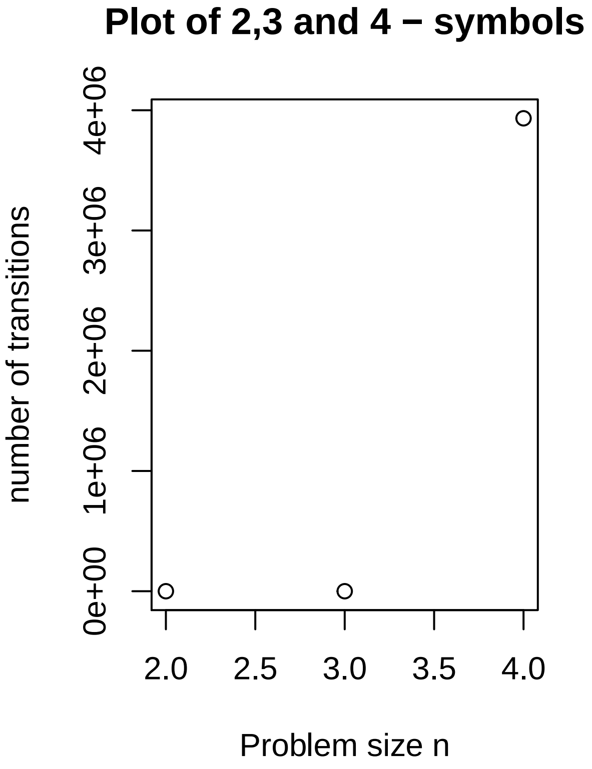
Experimental complexity analysis

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 anlysis. 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 x seconds. 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 in terms of transitions is roughly t, multiplying the time with the same factor gives t2. This would be until t3.

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. The function is also known to be uncomputational.

Summary

Non deterministic TM

Design

Problem 1

Theoretical Analysis

Experimental Analysis