Introduction :

The aim of this practical is to understand and program within the Turing machine model of computation. So what are Turing machines ?
A Turing machine is a mathematical model of computation. I has a head that reads symbols on a tape according to a table of rules. It is a very simple model that is capable of simulate any computer algorithm.

The practical is divided in 3 main tasks and an extension which is not attempted here. Thus, I am splitting our design and implementation in 2 distinct part as I explain my Turing machine simulator and the 4 machines created in task 2. I will then analyse, both theoretically and experimentally the complexity of these TM algorithms and TM simulation.

Design and Implementation :

Task 1:

The aim of this task is to construct a Turing machine simulator that takes a one tape deterministic machine and a half infinite tape. A lot of requirements are given to ensure a standardized version of the simulator to allow the running of further tests.

The first thing that needs to be done is creating the required project structure to run the program. I am using Java as the development language, writing a makefile was not too challenging as well as writing a bash script. However, this was something new and learnt, so worth mentioning.

Then, about the Java TM Simulator itself (JTMS), It is composed of 3 classes.

The main class TM takes care of argument handling, file creating and processing/method calls. It is commented and the structure is quite clear and simple. If the TM description file and the tape are provided, then both files are processed in that respective order. Otherwise, if only the tape is provided, the only difference is that we are processing the empty character “_”.

The Description class and the Machine class work together to start dealing with the description file. Indeed, when the process() method is called on the description file, it opens a buffered reader and reads the input lines. This method focuses on doing input checks to find syntax errors in the provided file. While passing those conditionals, it creates a Machine object, and uses the various adding methods to add states, alphabet and transitions. The Machine class uses Hash-maps to store those and uses and Transition object to save the transition line {q0,s0,q1,s1,L/R}.

Furthermore, We have the run() method in the Machine class where the TM tape is processed according to the rule table just stored and defined. To best understand what is going on in the main loop of this tape, Here is a shot pseudo-code with the main concepts:

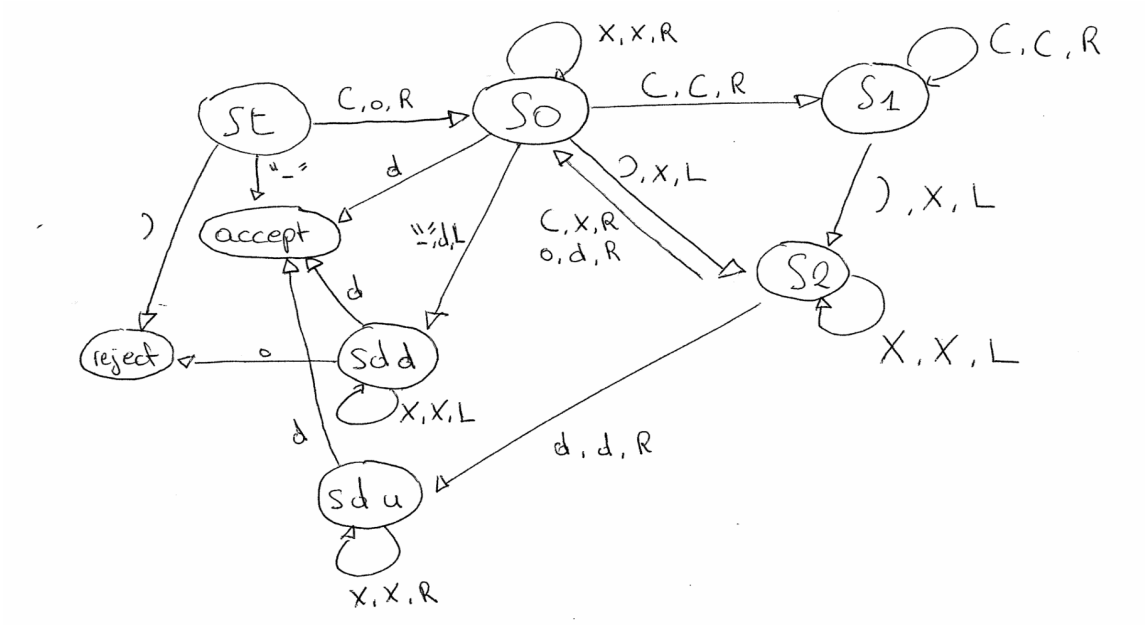
```
run ( input ){
        while(!accept && !reject){
                if (transSet(transition using current state and symbol at head) !=null){
                        we found a transition !
                        CurrentTransition =  the transition checked;
                }
                some checking;
                if (check passed){
                        tape to char[];
                        rewrite symbol at tapeChar[index of head];
                        move head depending on L or R;
                }
        }
}
```

Running the file processing and the tape run method, We get a viable working JTMS.

Task 2 :

This task is divided in 4 smaller tasks:

The parentheses balance checker is a TM machine that checks is the parentheses are balanced in a string. The idea behind it is that it first starts in a starting state “st” where the first ‘(‘ is marked as o. Then it moves right is search of a ‘)’ skipping all the ‘(’. It finds it, marks as X and moves left in search of ‘(’ or ‘o’ to mark them X or d respectively. It skips the X’s and continues the algorithm. When hitting the ‘_’ empty character that designates the end of the tape, it comeback to the left hand side looking for a d to confirm the string Is empty. This state is important because it allows strings such at “(()” to be balanced because it will detect the first bracket marked o instead of just ignoring it. Therefore, We set the TM diagram as show in the TM diagram :



The binary add Turing machine was a way more challenging part as I had a lot more states and transitions to handle. The binary adder uses a recursive binary tree logic, the Base structure of our TM machine thus looks like this :

the tree has 2 levels since we are adding 2 binary number, It is based on these basic binary addition rules :

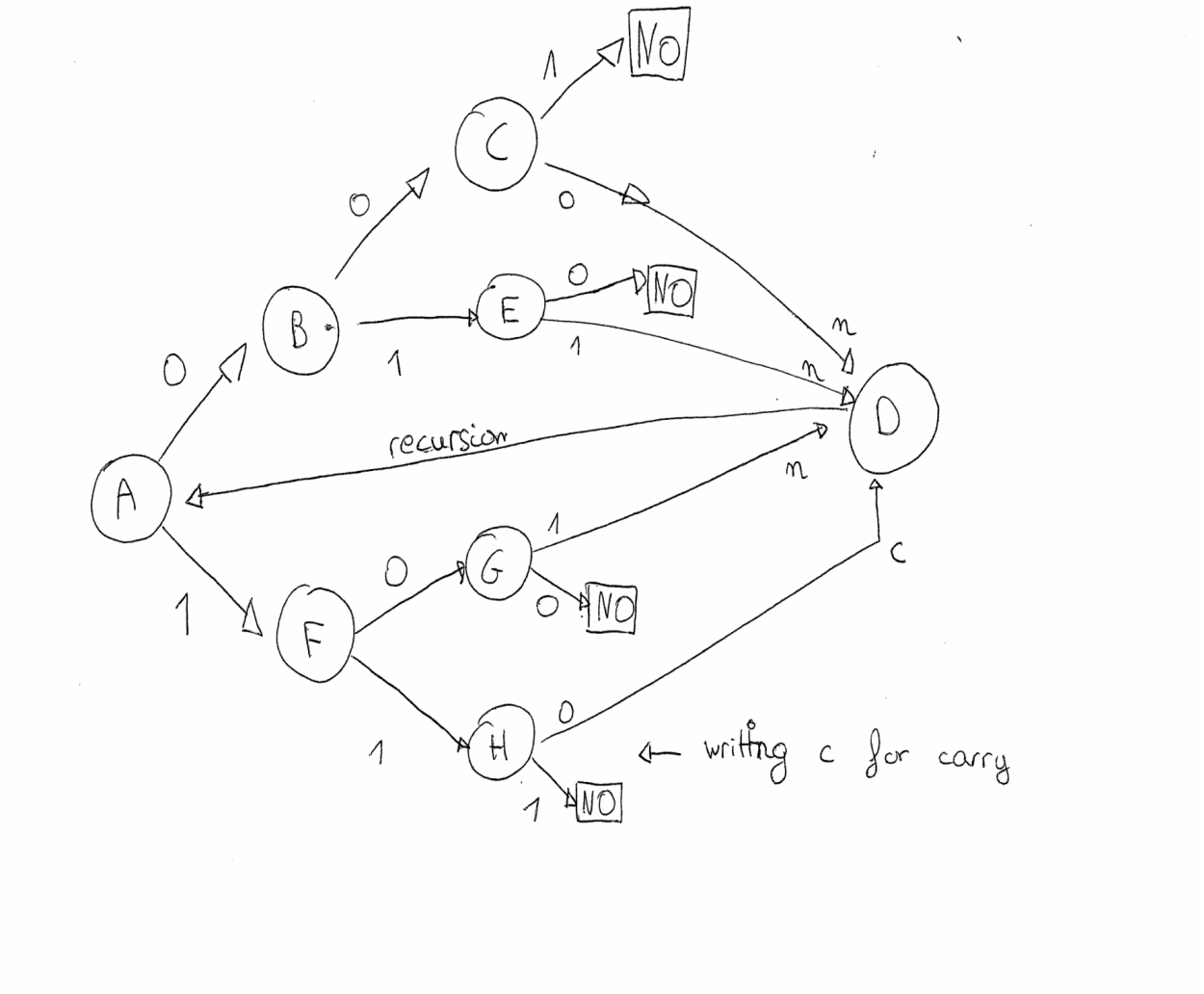
Case	A + B	Sum	Carry
1	0 + 0	0	0
2	0 + 1	1	0
3	1 + 0	1	0
4	1 + 1	0	1

If the Sum value is not the expected one, we halt to reject state and the tape is not accepted. However if the input is accepted, for the 3 first cases, we write ‘n’ as a non carry element and go to

the recursive state D that goes back to the left hand side (unmarked) to the initial state. In the case of case 4, we mark c to allow the reader to go in carry mode for further processing.
Every time we process a character, It is marked so that we can separate the first binary element from the second.

0#0#0 become pfdsn

{ p is a mark of a processed element from first binary number ;
f represents the first #, so the first separation ;
d is a mark of a processed element from second binary number ;
s is the second separation #;
n in non carry element ;
c is carry element; }

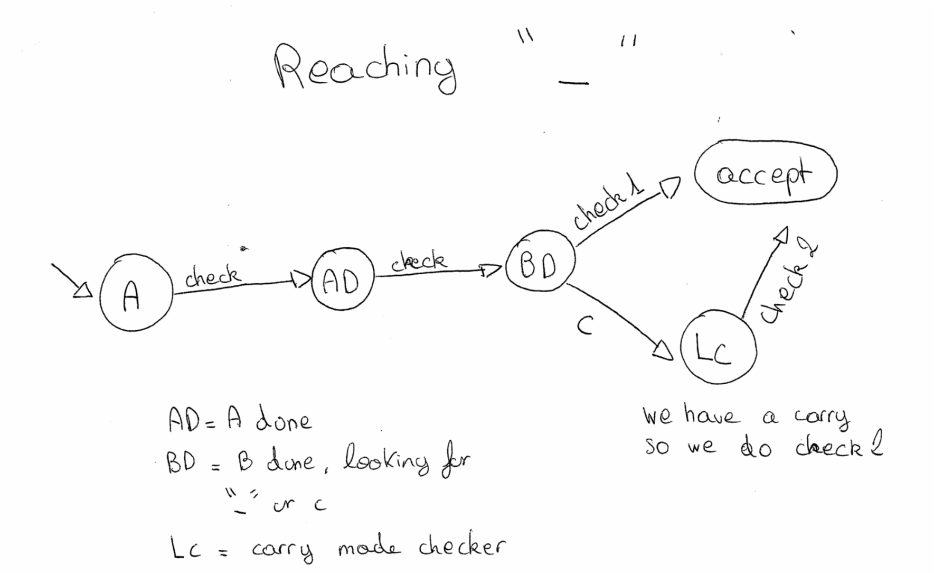


Pseudo-diagram of the Binadd TM

Now let's explain what is carry mode; when reading a c symbol on the tape, the TM machine goes into an alternate state that basically revert the expected table output such as :

Case	A + B	Sum	Carry
1c	0 + 0	1	0
2c	0 + 1	0	1
3c	1 + 0	0	1
4c	1 + 1	1	1

When the reader finally arrives at the "_" signalling the end of the tape, it comes back to the first element and iterates through the tape to check if everything is fine. Carry mode comes into play once again here. Indeed, the succession of n's and c's is check by the carry mode as well to allow a valid result. When reaching the end of the tape again, the algorithm goes to the accept state.
The final check takes place like so :



Pseudo-diagram of last check

The first original challenge is a binary palindrome checker and the second challenger is a divider by 3 checker. To best understand these TM machines, Let's use their TM transition table and use an example to comment on how it works.

Palindrome checker TM table

State / input	0	1	B
st	S0,B,R	S1,B,R	accept
s0	S0,0,R	S0,1,R	T0,B,L
s1	S1,0,R	S1,1,R	T1,B,L
t0	Td,B,L		accept
t1		Td,B,L	accept
td	Td,0,L	Td,1,L	St,B,R
accept			

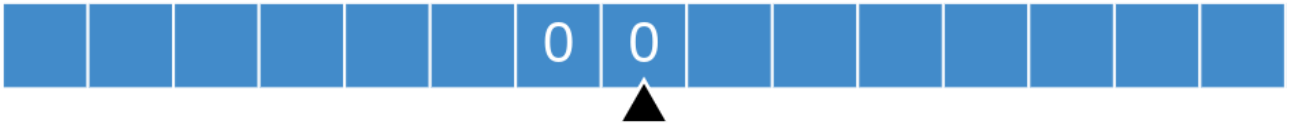
Let's take the binary "1001" and "011" :

we start at state "st"

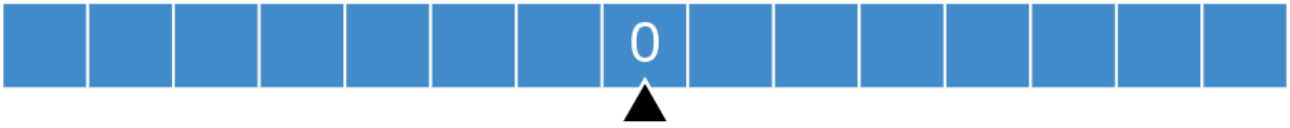
we have a 1 so we mark it blank and go right

we skip the other 0 until we hit the white space beside the one, We then come back to mark it as blank and keep moving left

we are on the 1 so we mark it blank



we are on the 0 so we mark it blank and move to the other blank space on the left hand side



we erase the 0 and move to the blank space on the right. We are in an accepting state so we have a palindrome.

Now using this algorithm on an input such as "011" would result in the tape :

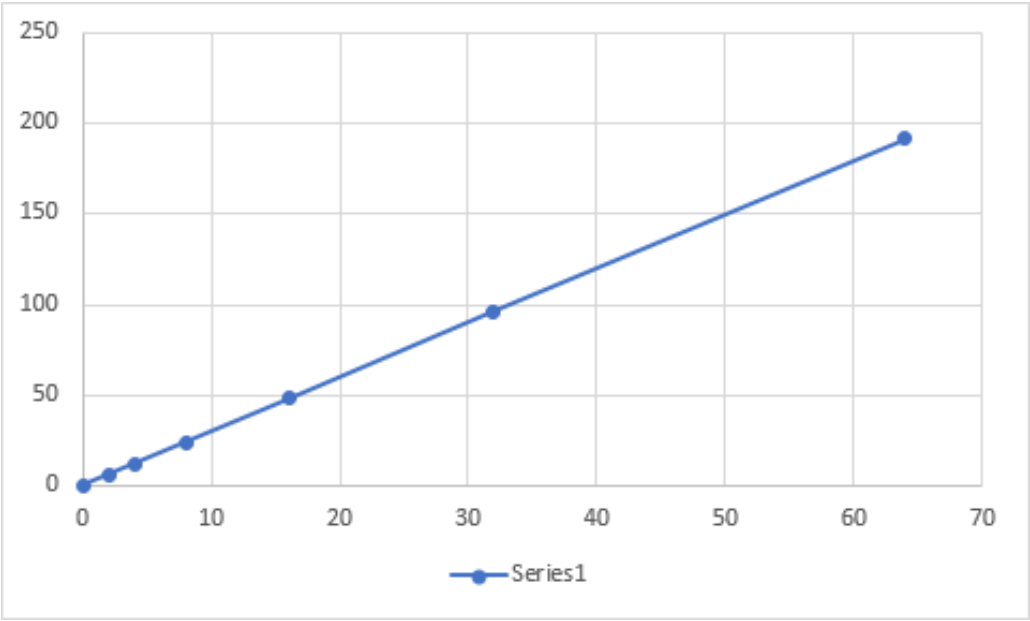
We are on a rejecting state after hitting the right blank and coming back on a 1.
Now concerning the second challenge, Here is the transition table :

State / input	0	1	B
S0	S0,0,R	S1,1,R	accept
S1	S2,0,R	S0,1,R	
s2	S1,0,R	S2,1,R	
accept			

This TM machine has 1 single accepting state, we don't need to use examples here since we can prove that it will always work using maths.
We can consider the 3 states as 3 different mod3 states:
s0 is mod3 == 0; s1 mod3 == 1 ; s2 mod3 == 2
when reaching the end of the tape on state s0, the binary number must be divisible by 3. Indeed, the algorithm basically keeps a count between 1's in odd positions and 1's in even positions. We know that a binary number is divisible by 3 if the difference of these counts Is equal to 0 mod3.
This is where the check takes place as s0 is mod3 == 0.

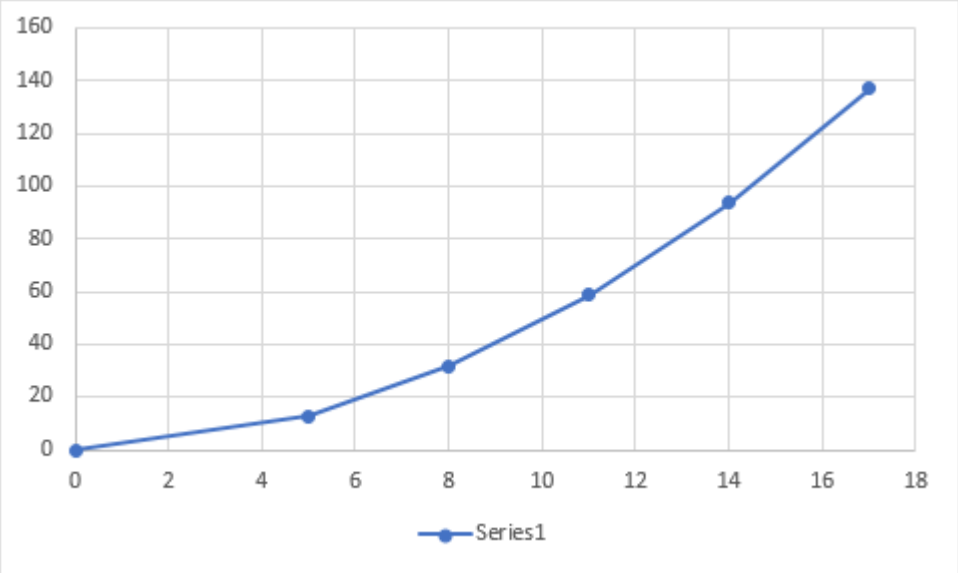
Complexity:

parentheses balance check complexity:



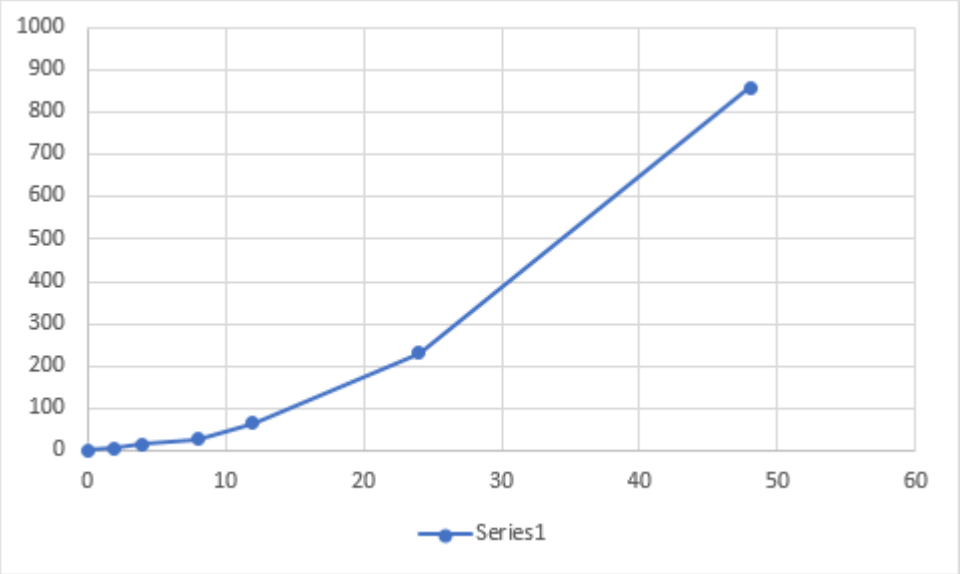
The algorithm loops through all the elements so It is over O(1) complexity. Furthermore, It doesn't loop through the input for every input. Thus, it is bellow O(n^2) complexity. We can understand easily from the algorithm itself and with just theory that we have a O(n) complexity. Now looking at the following graph above, We clearly see that it is a straight line which corresponds to a O(n) complexity.

Binary adder check complexity:



Since we have a recursive binary tree structure, the algorithm could look a bit like this :
for (input){traverse(tree)}
As we know, Both of these operations are O(n) since it iterates over the input. Being in the same form as : for(a){for(b){do stuff}}
we can assume that we have a O(n^2) complexity.
Moreover, we can clearly see that the curve we obtain from the input and steps ration is a half upward parabola which is characteristic of a O(n^2) complexity.

Palindrome checker complexity:



In the palindrome checker algorithm, we are recursively checking for that correct input for every character. The match seeking process makes the iteration over the input inevitable. Thus, we have a O(n^2) complexity. We could get a O(nlogn) complexity by recursively breaking the original problem into chunks each with a fraction of the size of the previous recursion.
The graph proves that we have a O(n^2) complexity as it is an upward parabola.

The division checker:
Concerning the division checker, the algorithm goes straight through the input without ever looping or recursing. Thus, we can be sure that we have a O(n) complexity.

Testing:

To test My Program, I mainly used the automated Checker. Here are the results of the different checks:

```
* TEST - test-ref-syn2 : pass
* TEST - test-ref-syn3 : pass
* TEST - test-ref-syn4 : pass
* TEST - test-ref-syn5 : pass
* TEST - test-sum1 : pass
* TEST - test-sum2 : pass
* TEST - test-sum3 : pass
* TEST - test-sum4 : pass
* TEST - test-syn1 : pass
* TEST - test-syn2 : pass
* TEST - test-syn3 : pass
* TEST - test-syn4 : pass
* TEST - test-syn5 : pass
38 out of 38 tests passed
```

```
* TEST - test-accept : pass
* TEST - test-busy5 : pass
* TEST - test-manystates : pass
* TEST - test-manysymbols : pass
* TEST - test-manytrans : pass
* TEST - test-manytrans2 : pass
* TEST - test-nonaccept : pass
7 out of 7 tests passed
mmd4@pc5-037-l:~/Documents/cs3052/Practical1/src $
```

```
mmd4@pc2-023-l:~/Documents/cs3052/Practical1/src $ stacscheck SimpleTM/
Testing CS3052 Practical 1 (Turing Machines)
- Looking for submission in a directory called 'src': Already in it!
* TEST - test-bad : pass
* TEST - test-blank : pass
* TEST - test-left : pass
* TEST - test-left2 : pass
* TEST - test-long : pass
* TEST - test-middle : pass
* TEST - test-notape : pass
* TEST - test-white : pass
8 out of 8 tests passed
```

These are the results for the Turing machine simulator Testing. I passed all the tests here, Although my program can be slow on the busy5 test but really fast in others.

```
* TEST - test-complex1 : pass
* TEST - test-complex2 : pass
* TEST - test-complex3 : pass
* TEST - test-complex4 : pass
* TEST - test-complex5 : pass
* TEST - test-empty : pass
* TEST - test-nest1 : pass
* TEST - test-nest2 : pass
* TEST - test-nest3 : pass
* TEST - test-open : pass
* TEST - test-ref-close-first : pass
* TEST - test-ref-close : pass
* TEST - test-ref-complex1 : pass
* TEST - test-ref-complex2 : pass
* TEST - test-ref-complex3 : pass
* TEST - test-ref-complex4 : pass
* TEST - test-ref-complex5 : pass
* TEST - test-ref-empty : pass
* TEST - test-ref-nest1 : pass
* TEST - test-ref-nest2 : pass
* TEST - test-ref-nest3 : pass
* TEST - test-ref-open : pass
24 out of 24 tests passed
```

```
* TEST - test-statenames1 : pass
* TEST - test-statenames2 : pass
* TEST - test-statenames3 : pass
* TEST - test-statenames4 : pass
* TEST - test-statenames5 : pass
* TEST - test-statenames6 : pass
27 out of 27 tests passed
```

These are the tests for my parentheses checker and my binary adder check. I passed all the test and made my TM's solid by adding states and conditions. I think that my binary adder is fairly performant and uses an original algorithm.

Conclusion:

This practical was very long to do and to test. Overall, I completed all basic requirements at a respectable level as I pass all the tests and used clever algorithms. I also made some mistakes that I fixed but that taught me a lot about complexity.