CS3052 Practical 1

160021428

12/03/2019

Overview

The aim of this practical is to create a program that can simulate a Turing Machine by parsing a description file containing the details and states of a Turing Machine. This will be tested against varying types of tapes which may be accepted or rejected by the machine. In addition, using the parser, a number of description files have been developed that allow the machine to accomplish a number of different tasks and take in different types of tapes. The resulting output of these Turing Machines will then be measured to evaluate the Time complexity of their algorithms.

Design

To create the Turing Machine Parser, Java was selected as the primary programming language as its object-oriented design allowed for quite a simple implementation and the necessary abstractions of certain aspects of the Turing Machine. As the design of a Turing Machine is quite simple, any creative implementations are very limited aside from how the data is stored and accessed during the tape processing stage.

ParenTM

ParenTM works properly by replacing the initial character with the letter “s” to signify the start of the tape, then runs along the tape until it reaches the other end where it starts scanning for any opening brackets from right to left. Once it hits an open bracket, it changes state and goes right until it hits a closing bracket (**Appendix A**), signifying that there is at least one balanced bracket in the tape. The TM then repeats the process until it clears the tape of all brackets which determines whether the tape is balanced or not. At any point during the scanning process, if the tape hits the blank character at the right end of the tape, it is determined as unbalanced and therefore rejected.

BinAddTM

In contrast to ParenTM, the design of the Binary Addition Turing Machine is slightly more complicated as it requires quite a bit more states and quite a bite more rules related to them. However, it is designed in a similar manner where the initial character is marked as a special character that allows the machine to find its way back to it. To begin the check, the value of the initial number is taken and added to the value of the second number at the same index. For instance, if the first number is 101 and the second number 011, then the values to be added will be 1 and 0 as they are the values at the first index of their corresponding number (**Appendix B**). Once the sum of both values has been attained, the value at the same index of the final number will be compared to the sum. If the value is valid, the number is crossed out and the tm will go back to the beginning of the tape to repeat the same checking process until the values of all three numbers are crossed out.

This implementation works perfectly fine until there are is an addition sequence wherein a consecutive number of 1's needs to be carried over to the value of the next index. To deal with this, a new character "t", and state "carryOne", has been implemented to signify and look out for a carry one (**Appendix C**). When a 1 is added to another 1 in the second number, instead of crossing the 1 out, a t will be placed in its stead. This will be used to signify that the value is a temporary value and will be added in during the next cycle. After setting the temporary value, the Turing Machine can continue normally. Once the TM returns back to the t in the next state, it will be treated as another 1 and be added to the total sum of the current index.

In addition, the TM can deal with an empty/zero first number by changing the starter marker from an s to a z instead (**Appendix D**). This signifies that the first number is zero and will therefore add zero to every index of the second number. However, this does not need to be implemented to the second number as the proper format for a tape of this TM requires two hashtags which allow it to recognize when a change of state needs to be made.

PalinTM

One of the custom Turing Machines that was implemented, is a TM that can check if a string consisting of X and Y characters is a proper palindrome. Its implementation uses a similar method to the Balanced Parenthesis Turing Machine and goes across the tape crossing out matching characters at the beginning and end of the tape. This is done by checking if the first non-empty character (empty characters are signified by a -), at the start of the tape has a matching counterpart at the end of the tape. This is done repeatedly until the tape is accepted, wherein there are no more characters on the tape aside from the start symbol s (**Appendix E**). If the first non-empty character at the beginning of the tape does not have a matching counterpart at the end of the tape, the tape is rejected.

MakePalinTM

The second custom Turing Machine is one that creates a palindrome from a string of X and Y characters. This allows the previous PalinTM to check the validity of the palindrome and vice versa. To successfully implement this, the TM pastes a mirrored copy of the current tape at its end to create a palindrome. It does this by reading the initial character and setting it to either 1 or 0 to represent X and Y respectively, but it is also used to signify the position of the starting character. After reading and marking the starting character, the machine will go to the end of the tape and copy and paste the last character to the end of the tape. A character that has been copied, aside from the initial character, is marked as x or y. Once appended to the end of the tape, the TM will return to the end of the original tape and look for the first viable character (**Appendix F**). This process is repeated until the machine copies the starting character to the end of the tape. After appending the final character, every marked character (e.g. x, y, 1, and 0) will be reverted to their original state, thus, leaving a palindrome sequence.

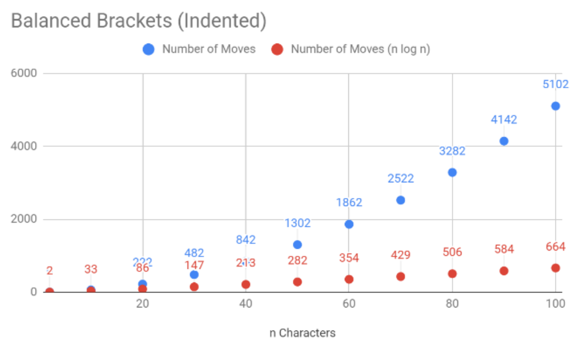
Implementation

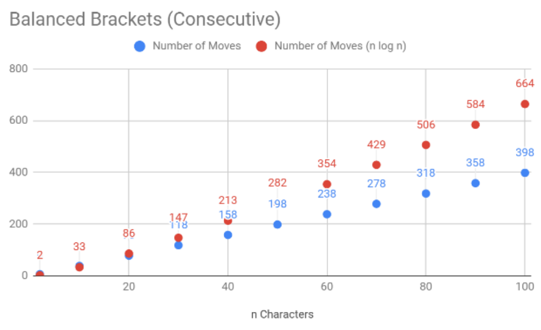
In reference to the design section in regarding the method of storing and accessing data, a HashMap was used to store the individual states to reduce any transition time that may be caused by the machine rather than the efficiency of the TM description. This allows the program to be as close to an actual Turing Machine and have each transition have a time complexity as close to O(1). In contrast, the tape and alphabet are both represented as Lists as the values they hold are meant to be accessed sequentially as per the design of a Turing Machine. This is especially true for the tape as only one character needs to be accessed and changed at a time, with the tape then moving right or left depending on the instruction of the transition.

Testing

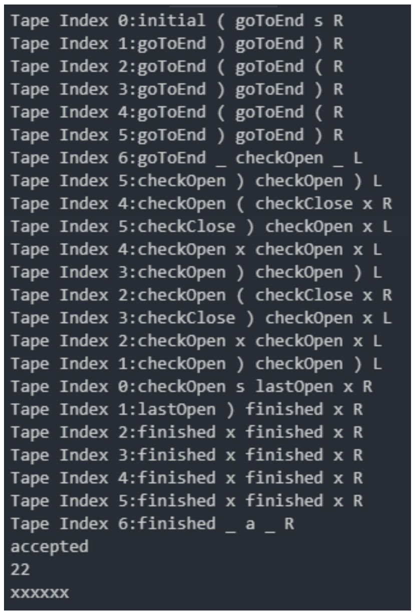
To properly show the efficiency of each Turing Machine for a tape of length n, each of the machines are given an input tape wherein n is increments of 10 until 100. With the given input the complexity of each machine can then be properly analyzed. Included in each of the graphs (red) is O(n log n) that allows for a comparison between the two complexities. Although the best-case scenario of an algorithm is not generally taken into consideration as the best case of most algorithms is O(1) (constant time), measuring the best case performance of a Turing Machine may provide more insight to its efficiency.

Balanced Brackets (ParenTM)



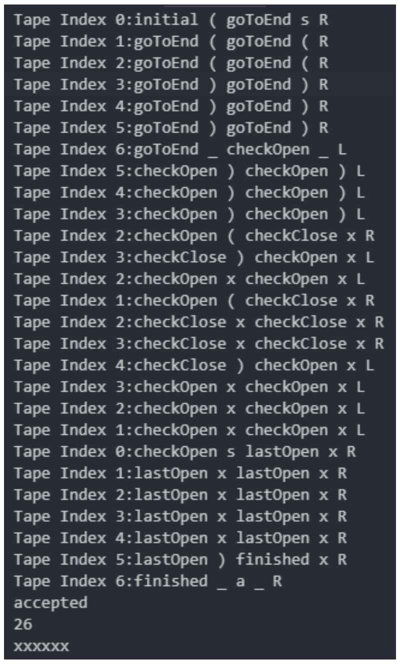


The design of the ParenTM machine has been tested with two different kinds of inputs as it was noticed that its efficiency varies depending on the type and order of its input parentheses. The two different inputs are Indented Balanced Brackets which take the form of "((()))", while Consecutive Balanced Brackets are parentheses laid out as "()()()" next to each other. As can be seen from the Indented Balanced Brackets graph, the time complexity of the machine tends towards O(n^2) which is the typical complexity of most Turing Machines. However, using consecutive balanced brackets as an input it appears that the number of moves taken by the machine is significantly less and effectively, more efficient than O(n log n). When each transition is printed out it can be clearly seen as to why this is the case.



```
Tape Index 0:initial ( goToEnd s R
Tape Index 1:goToEnd ) goToEnd ) R
Tape Index 2:goToEnd ( goToEnd ( R
Tape Index 3:goToEnd ) goToEnd ) R
Tape Index 4:goToEnd ( goToEnd ( R
Tape Index 5:goToEnd ) goToEnd ) R
Tape Index 6:goToEnd _ checkOpen _ L
Tape Index 5:checkOpen ) checkOpen ) L
Tape Index 4:checkOpen ( checkClose x R
Tape Index 5:checkClose ) checkOpen x L
Tape Index 4:checkOpen x checkOpen x L
Tape Index 3:checkOpen ) checkOpen ) L
Tape Index 2:checkOpen ( checkClose x R
Tape Index 3:checkClose ) checkOpen x L
Tape Index 2:checkOpen x checkOpen x L
Tape Index 1:checkOpen ) checkOpen ) L
Tape Index 0:checkOpen s lastOpen x R
Tape Index 1:lastOpen ) finished x R
Tape Index 2:finished x finished x R
Tape Index 3:finished x finished x R
Tape Index 4:finished x finished x R
Tape Index 5:finished x finished x R
Tape Index 6:finished _ a _ R
accepted
22
xxxxxx
```

Figure A: Consecutive Example (Balanced Brackets)

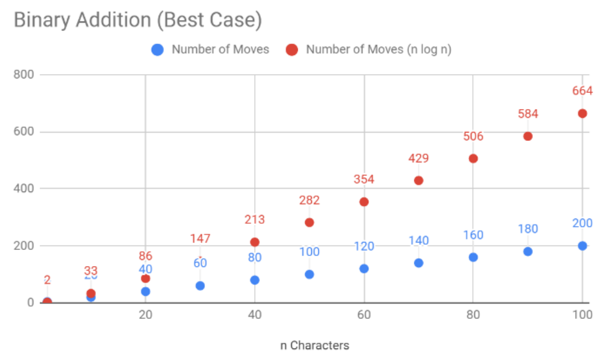


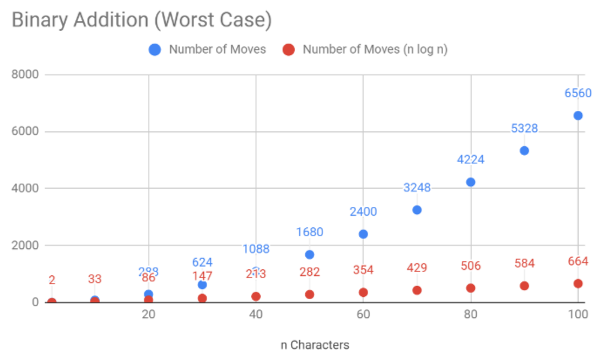
```
Tape Index 0:initial ( goToEnd s R
Tape Index 1:goToEnd ( goToEnd ( R
Tape Index 2:goToEnd ( goToEnd ( R
Tape Index 3:goToEnd ) goToEnd ) R
Tape Index 4:goToEnd ) goToEnd ) R
Tape Index 5:goToEnd ) goToEnd ) R
Tape Index 6:goToEnd _ checkOpen _ L
Tape Index 5:checkOpen ) checkOpen ) L
Tape Index 4:checkOpen ) checkOpen ) L
Tape Index 3:checkOpen ) checkOpen ) L
Tape Index 2:checkOpen ( checkClose x R
Tape Index 3:checkClose ) checkOpen x L
Tape Index 2:checkOpen x checkOpen x L
Tape Index 1:checkOpen ( checkClose x R
Tape Index 2:checkClose x checkClose x R
Tape Index 3:checkClose x checkClose x R
Tape Index 4:checkClose ) checkOpen x L
Tape Index 3:checkOpen x checkOpen x L
Tape Index 2:checkOpen x checkOpen x L
Tape Index 1:checkOpen x checkOpen x L
Tape Index 0:checkOpen s lastOpen x R
Tape Index 1:lastOpen x lastOpen x R
Tape Index 2:lastOpen x lastOpen x R
Tape Index 3:lastOpen x lastOpen x R
Tape Index 4:lastOpen x lastOpen x R
Tape Index 5:lastOpen ) finished x R
Tape Index 6:finished _ a _ R
accepted
26
xxxxxx
```

Figure B: Indented Example (Balanced Brackets)

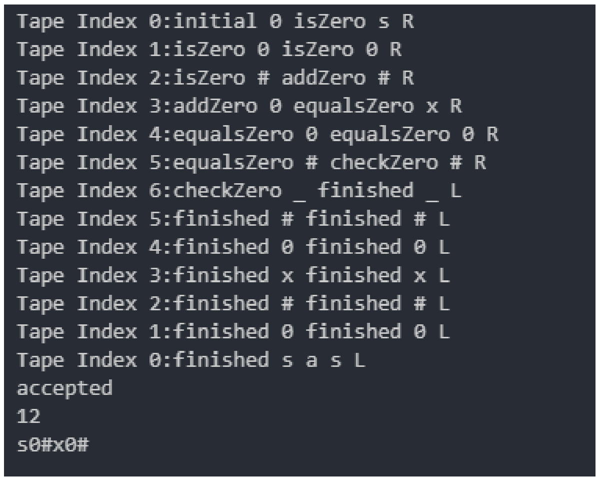
As can be seen in figure B, TM must traverse a longer distance for the Indented brackets once the innermost brackets have been matched. The number of unproductive moves (moves that don’t change any value), increments by 1 whenever the TM attempts to match an open bracket to a closing bracket as well as when a new open bracket is to be found. This is because it must still traverse over all the empty cells. In contrast, since the open and closing parentheses of Figure A are ordered consecutively, the number of moves required to move to another set of parentheses consistently stays at 3.

Binary Addition (BinAddTM)





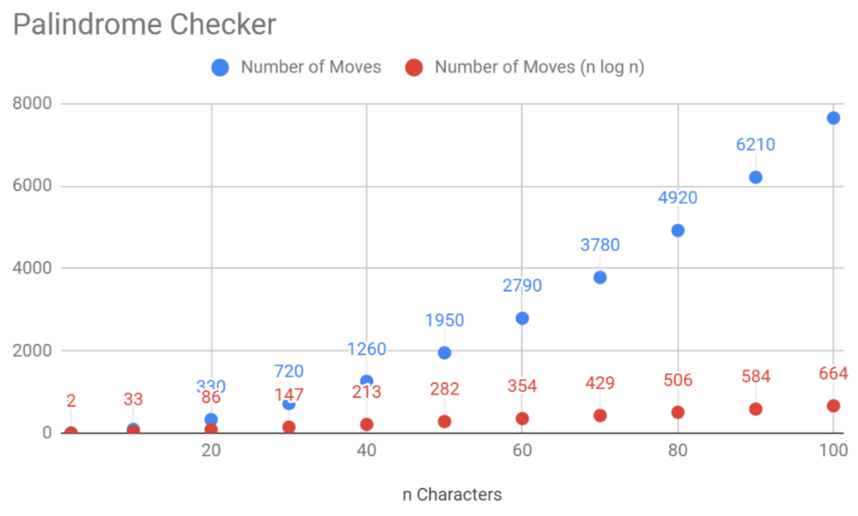
Although the worst case of the BinAddTM has a similar complexity to ParenTM which tends to n^2, its best-case complexity is 2n. To achieve the best case, the tape needs to have the first and second number as a set of 0s while the sum is left blank (e.g. 00#00#). This is the case due to the design of the machine, which as stated in its design, adds the value at the first index of the first number to the first index of the second number, then checks the last number for the sum at the same index. However, if at the final number, the blank character is reached and the sum of the indices of the first two numbers is equal to zero, then the summation is assumed to be finished. This causes the machine to change into a finished state which backtracks all the way to the initial character. If the machine reaches the initial character without coming across any 1s, then the tape is accepted. Hence, the best-case complexity of the machine is 2n as can be seen in Figure C, with an input tape of 00#00#.



```
Tape Index 0:initial 0 isZero s R
Tape Index 1:isZero 0 isZero 0 R
Tape Index 2:isZero # addZero # R
Tape Index 3:addZero 0 equalsZero x R
Tape Index 4:equalsZero 0 equalsZero 0 R
Tape Index 5:equalsZero # checkZero # R
Tape Index 6:checkZero _ finished _ L
Tape Index 5:finished # finished # L
Tape Index 4:finished 0 finished 0 L
Tape Index 3:finished x finished x L
Tape Index 2:finished # finished # L
Tape Index 1:finished 0 finished 0 L
Tape Index 0:finished s a s L
accepted
12
s0#x0#
```

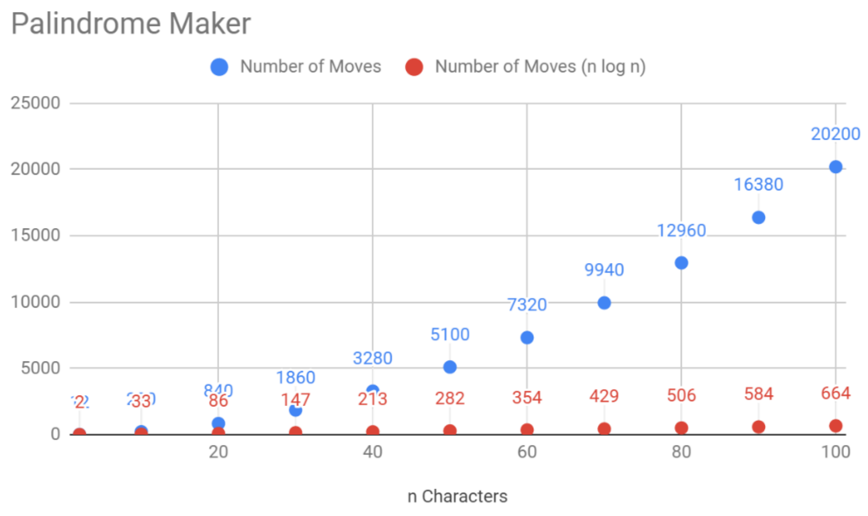
Figure C: Best Case example (Binary Addition)

Palindrome Checker (PalinTM)



In contrast to the complexity of both the Balanced Brackets and Binary addition machines, the Palindrome checker has a consistent Time complexity of O(n^2) regardless of its input tape. This is because it is designed to check for a character from one end of the tape and go to the other end to see if the same character exists there. It will continue to follow the same process regardless of the input character, thus causing its time complexity to be the same for all cases.

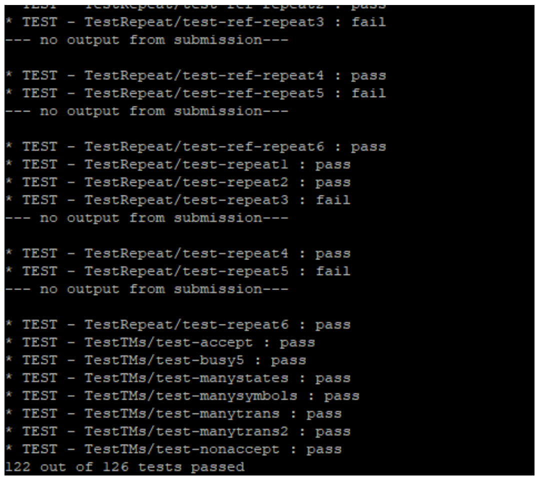
Palindrome Maker (MakePalinTM)



Like the Palindrome Checker, the Maker also has a consistent time-complexity that tends to O(n^2) as it goes to the end of the tape and copies and pastes each character in a mirrored sequence before returning to the beginning of the tape to check if all characters have successfully been copied. Although its complexity tends to O(n^2), that is not its exact complexity. Its exact complexity can be solved using the formula (n^2 * 2) + 2n, where n is the length of the tape.

Extension

The first part of the extension was completed and successfully passes all of the Non-Deterministic tests, however, the second part of the extension which requires a repeat.tm, has only been partially completed. It passes a few tests but fails in 4 different tests (Figure D).



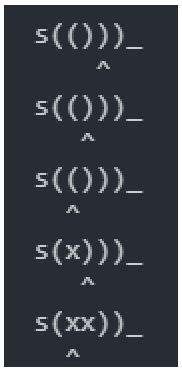
```
* TEST - TestRepeat/test-ref-repeat3 : fail
--- no output from submission---

* TEST - TestRepeat/test-ref-repeat4 : pass
* TEST - TestRepeat/test-ref-repeat5 : fail
--- no output from submission---

* TEST - TestRepeat/test-ref-repeat6 : pass
* TEST - TestRepeat/test-repeat1 : pass
* TEST - TestRepeat/test-repeat2 : pass
* TEST - TestRepeat/test-repeat3 : fail
--- no output from submission---

* TEST - TestRepeat/test-repeat4 : pass
* TEST - TestRepeat/test-repeat5 : fail
--- no output from submission---

* TEST - TestRepeat/test-repeat6 : pass
* TEST - TestTMs/test-accept : pass
* TEST - TestTMs/test-busy5 : pass
* TEST - TestTMs/test-manystates : pass
* TEST - TestTMs/test-manysymbols : pass
* TEST - TestTMs/test-manytrans : pass
* TEST - TestTMs/test-manytrans2 : pass
* TEST - TestTMs/test-nonaccept : pass
122 out of 126 tests passed
```

Figure D: Number of tests passed.

Conclusion

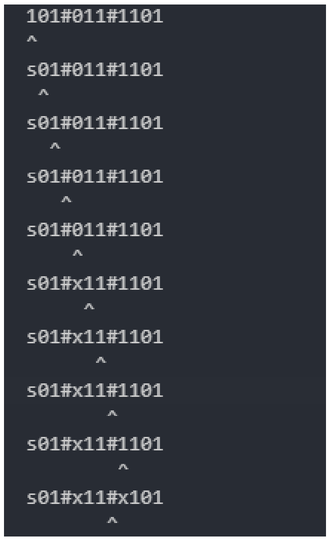
To conclude, during the duration and development of the practical, I was able to successfully implement most of the required Turing Machines and a few of my own (aside from the extension). This shows implies a better understanding of the subject at hand and Turing Machines in general. However, as can be seen in some of the experiments above, there are some aspects that can definitely be improved upon, especially in relation to the subject of Non-Deterministic Turing Machines.

Appendix



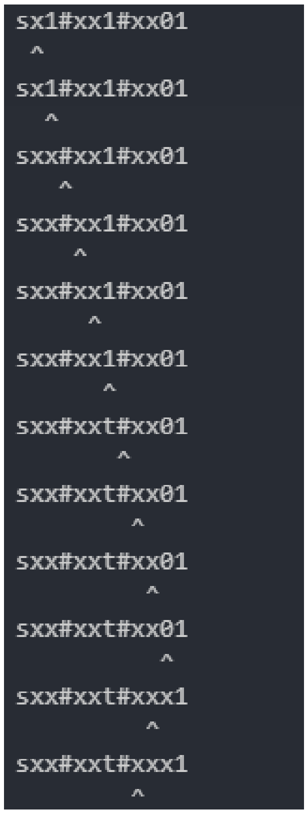
```
s(()))_
    ^
s(()))_
   ^
s(()))_
  ^
s(x)))_
   ^
s(xx))_
  ^
```

Appendix A: Cancelling (Balanced Brackets)



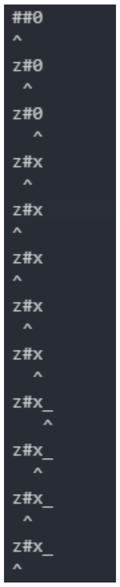
```
101#011#1101
^
s01#011#1101
 ^
s01#011#1101
  ^
s01#011#1101
   ^
s01#011#1101
    ^
s01#x11#1101
     ^
s01#x11#1101
      ^
s01#x11#1101
       ^
s01#x11#1101
        ^
s01#x11#x101
       ^
```

Appendix B: Adding values (Binary Addition)



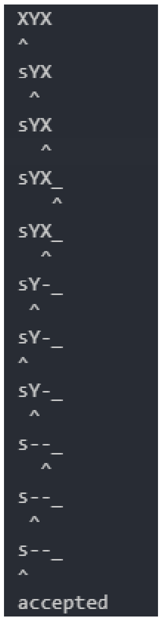
```
sx1#xx1#xx01
 ^
sx1#xx1#xx01
  ^
sxx#xx1#xx01
   ^
sxx#xx1#xx01
    ^
sxx#xx1#xx01
     ^
sxx#xx1#xx01
      ^
sxx#xxt#xx01
       ^
sxx#xxt#xx01
        ^
sxx#xxt#xx01
         ^
sxx#xxt#xx01
          ^
sxx#xxt#xxx1
         ^
sxx#xxt#xxx1
        ^
```

Appendix C: Carry one (Binary Addition)



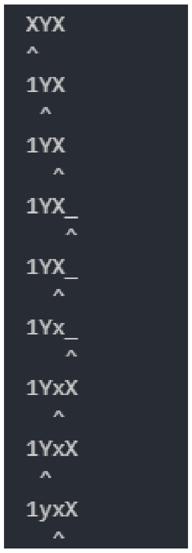
```
##0
^
z#0
 ^
z#0
  ^
z#x
 ^
z#x
^
z#x
^
z#x
 ^
z#x
  ^
z#x_
   ^
z#x_
  ^
z#x_
 ^
z#x_
^
```

Appendix D: Empty values (Binary Addition)



```
XYX
^
sYX
 ^
sYX
  ^
sYX_
   ^
sYX_
  ^
sY-_
 ^
sY-_
^
sY-_
 ^
s--_
  ^
s--_
 ^
s--_
^
accepted
```

Appendix E: Accepted Tape (Palindrome Checker)



```
XYX
^
1YX
 ^
1YX
  ^
1YX_
   ^
1YX_
  ^
1Yx_
   ^
1YxX
  ^
1YxX
 ^
1yxX
  ^
```

Appendix F: Copy and Pasting (Palindrome Maker)