Turing Machines

David Kennedy

February 2023

Contents

1 Introduction

In this practical assignment, we were to develop our understanding of the Turing Machine abstraction by building a TM simulator, and also designing a number of Turing Machines for various tasks. Furthermore, we were tasked with analysing the complexity of our TM algorithms, either theoretically or experimentally (or both) .

I designed and developed my TM simulator in Python, developed Turing Machines to solve the two problems detailed in the specification, and specified and solved two additional problems. All of my Turing Machine solutions to the specified problems pass the provided StacsCheck tests (**REDACTED**), and I performed additional tests on my solutions to the auxiliary problems. Each TM implementation has been analysed in terms of complexity, with the experimental results confirming the theoretical expectations.

2 Design

2.1 Turing Machine Simulator

Unlike previous practical assignments, this time we were given the choice as to which programming language to implement our simulator in, as the focus should be on developing our understanding of Turing Machines, not a specific language. Naturally, I attempted to decide between the two languages I am most familiar with - Java and Python. While Java is undoubtedly faster as a result of its compilation, it has stricter typing and requires more boilerplate code during development in comparison to Python; as Python is, on average, easier to understand and faster to develop in than Java, I decided to use Python to prototype the TM simulator. If speed became an apparent issue during testing, I would have then switched to Java and rewritten the simulator using the prototype as a blueprint - however, my Python implementation was able to narrowly pass the speed requirements given in the specification, as shown in the Testing section, and there was no need to recreate the program.

When analysing the requirements for the Turing Machine simulator, it was apparent that computation could be split into three main sections: reading the TM description file, reading the tape, and then running the simulation. I therefore began development by splitting my program into three separate functions, as well as a main function, in order to both split the code into logical blocks, and improve ease of readability and understanding. This process was then repeated once more, as I found that reading the TM description file again contained three parts: checking the states, documenting the alphabet, and recording the transitions. By splitting my program into several functions, not only was I able to improve my debugging experience later on in development, I also enhanced the clarity surrounding the data-flow throughout the simulator, allowing developers unfamiliar with my program to recognise how information is transferred throughout the simulation process.

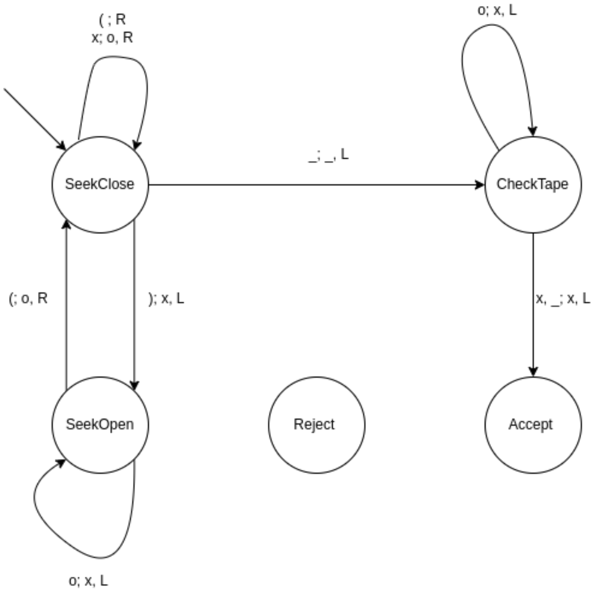
Due to Python's interpreted execution, it can become quite slow when dealing with large numbers of commands, compared to its compiled counterparts; for this reason, I knew my use of data types and structures had to be extremely calculated in order to reach an acceptable performance time. Throughout my implementation, I made sure to store any information that did not need to be ordered in sets - on average, sets provide a lookup, insert, and delete time complexity of $O(1)$ [1]. Being able to check if transitions were valid or if a character used within a tape was permitted by the given alphabet in constant time was of massive importance to the speed of my implementation, especially when dealing with larger datasets; for example, if I had used lists, the average lookup time complexity would have jumped to $O(n)$. I made sure to store all available states in a set using the tuple format (state name, status), as there was mostly no need to ever sort states. The only case where indexing would have been useful was documenting the initial state, but this was easily solved by creating a variable to store the first state read from the TM description file. Of course, certain data could not be stored in sets - the order of the tape passed read by the Turing Machine is incredibly important, and the data structure containing it should be iterable.

Another data structure I used extremely effectively was Python's 'dictionary', or hash table. A hash table stores key-value pairs, allowing retrieval of values using the respective keys - insert, retrieval, deletion, and checking if a pair is in a dictionary all have an average time complexity of $O(1)$, constant time [2]. Storing transitions between states within a hash table using the format *(state, tape) → (state, tape, direction)* allowed my program to check for transitions and retrieve corresponding instructions in constant time, leading to an incredibly efficient implementation.

The specification lists a number of precise criteria that make up a valid TM description file and tape, as well as individual exit statuses - to follow these strict rules, I employed the use of assert statements and try-except blocks. Implementing assert statements within my program allowed me to perform a number of checks regarding the TM description file and tape - if one of these assertions were not true, then an exception would be thrown. Using the exception type specified by the assert statement, as well as the fact I had split my code into logical operational functions, I was able to use the except block to return the correct exit status once the program had stopped, satisfying the specification's requirements.

2.2 Specified Problems - Parentheses

Recognise the language of balanced bracket sequences. That is sequences in (,) * with equal numbers of opening and closing parentheses, such that every opening parenthesis can be matched with a closing parenthesis that occurs after it. So , (), (()) and ()() are all in the language, but (,)(and ((()(are not.

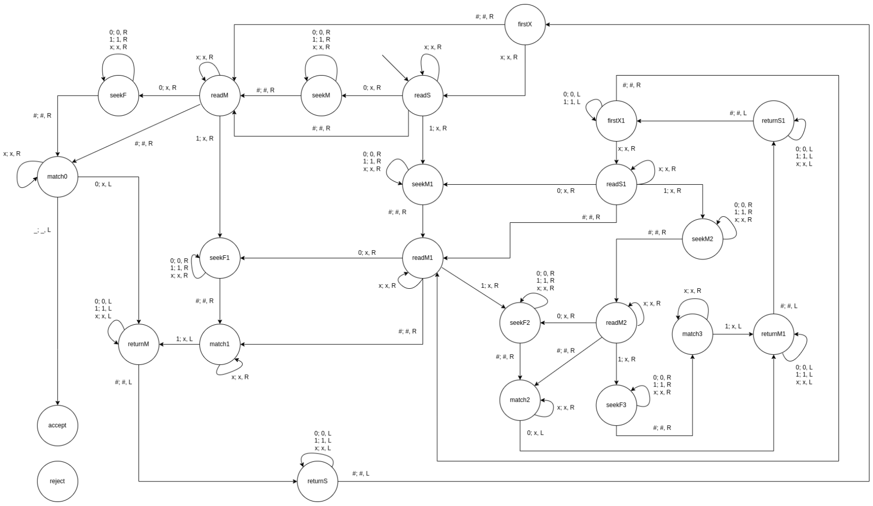


To solve the problem of recognising the language of balanced bracket sequences, I used a total of five states, and a four-letter alphabet. Once initiated, the algorithm immediately starts searching for a ‘close’ bracket ‘)’, ignoring all other characters on the tape. The first close bracket it finds is then replaced with an ‘x’, and the search for the corresponding open bracket begins. Once the bracket is found, it is replaced with an ‘o’, and the process continues indefinitely, with the seekClose state replacing ‘x’s with ‘o’s, and the seekOpen state doing the opposite.

Eventually, provided that the tape contains a valid sequence, a blank space on the tape will be reached - at this point, the algorithm will check over the tape, expecting to find every position of the tape containing an ‘o’, and converting each ‘o’ into an ‘x’. Once the beginning of the tape is reached, the checkTape state will convert the initial ‘o’ to an ‘x’, attempt to move left but stay in the same position, and then read ‘x’, at which point the TM will move into the accept state.

2.3 Specified Problems - Binary Addition

> Recognise strings from the language w1#w2#w3, where w1, w2, and w3 are binary numbers, least significant bit first, and w1 + w2 = w3 in binary. For instance, 0#0#0, 01#1#11, and 00#111#111 belong to the language, but 0#0, 0#0#1, and 000#111#11 do not.

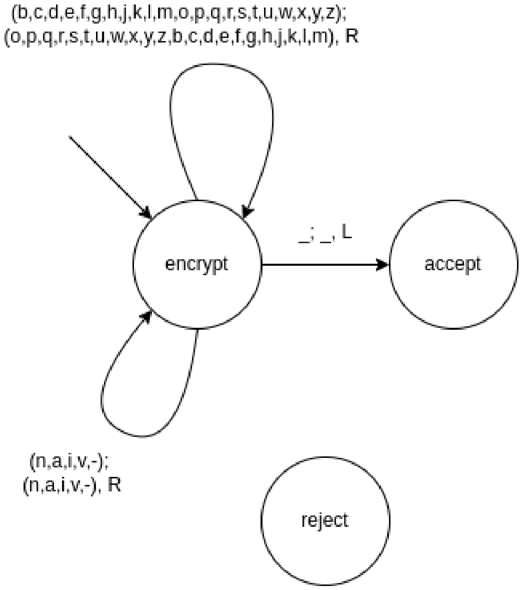


The implementation of my solution to this problem required 24 states, and a four-character alphabet. The larger number of states allowed my TM to keep track of any carry bits that may have resulted from the previous addition, the maximum being three. Once initialised, my algorithm would read the first bit of the first string, replace it with an 'x', and depending on the value of the bit, move into one of two states: seekM, or seekM1 (1 signifying the carry bit). This process would be repeated once more for the middle binary string, after which the read head would move to the last binary string, and check the number of carry bits against the bit read. If these two values correlate, then the read head would move back to the initial binary string using the '#'s as indicators, locate the last-read bit in the string, and then move right to read the next one. The TM only reaches the accept state when the entire tape contains only 'x's and '#'s - the algorithm should reach the match state with zero carry bits, eventually read a blank character, and then move to the accept state. Therefore, it is not possible to reach the accept state if one or more carry bits have not been resolved.

In retrospect, using '#'s as indicators to guide the journey back to the start of the tape seems unnecessary, and an inefficient use of states - by increasing the size of the alphabet by 2 characters, it would be possible for each binary string section to have its own 'read' character, facilitating the elimination of the 'seekM' and 'seekS' states, reducing the total number of states to 20.

2.4 Additional Problems - NAIV cipher

> Recognise strings containing letters from the lowercase English alphabet and '-', and encrypt these strings on-tape using a substitution cipher with shift-size 13; the letters 'n', 'a', 'i' ,'v' should not be shifted. The '-' should also not be shifted, as it is to be a placeholder for a space between words.


(b,c,d,e,f,g,h,j,k,l,m,o,p,q,r,s,t,u,w,x,y,z);
(o,p,q,r,s,t,u,w,x,y,z,b,c,d,e,f,g,h,j,k,l,m), R
encrypt
_; _, L
accept
(n,a,i,v,-);
(n,a,i,v,-), R
reject

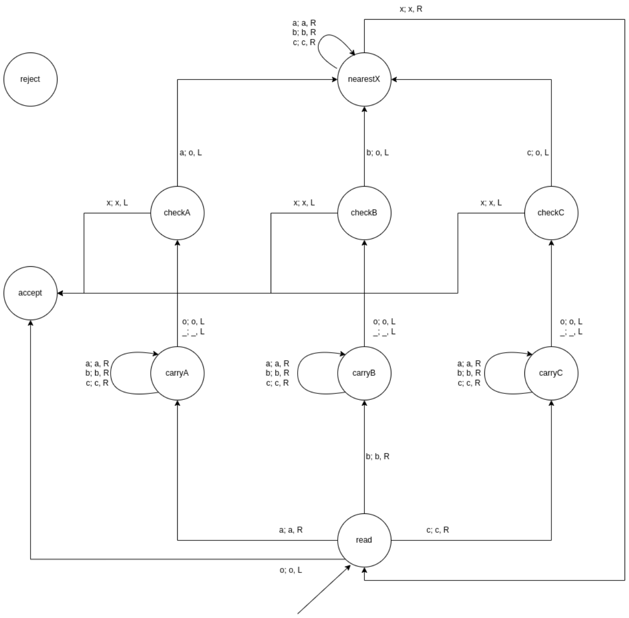
Initially, I thought it would be fun to make a simple symmetric Caesar cipher; to make light of the simplicity of this encryption technique, I left out the letters 'n, a, i, v' (naive) from the shifting process, as they are 13 positions apart respectively. The Turing Machine should accept a tape containing any standard lowercase letter from the alphabet, along with hyphens to act as spaces, and reach the accept state once each character is shifted as defined; the resulting tape should contain only the encrypted version of the original message. However, after creating a TM to solve this problem I realised that while it was a fun problem to solve, it likely didn't qualify as challenging - the lack of complexity of my solution is clearly shown in the diagram below, and any tape containing characters from the alphabet would be accepted.

The 'encrypt' state contains a single transition for every letter in the alphabet - it shifts every character on the tape until it reaches the end of the tape, i.e., '_', at which point it enters the accept state.

Despite it being a very simple problem to solve, I thought I would include it anyway, as I found the implementation and usage to be quite fun.

2.5 Additional Problems - Palindrome

Recognise the language of Palindromic sequences; that is, odd or even sequences in a, b, c+ that read the same forwards or backwards. So ‘a’, ‘aa’, ‘aba’, and ‘abcba’ are all in the language, but ‘ab’, ‘abc’, and ‘abab’ are not.



Compared to the previous cipher, this problem is a considerable step up in complexity - while admittedly it is somewhat similar to the parentheses problem, Palindromic sequences can be odd or even, unlike the solely even parenthetical sequences, introducing some added complexity.

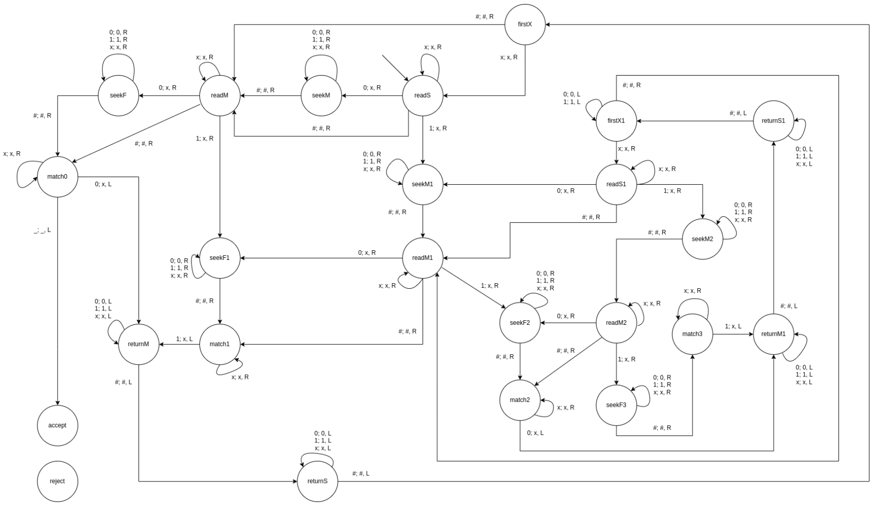
On initialisation, the TM reads the first character on the tape, replaces it with an ‘x’, and moves the end of the tape - once the end is reached, it moves left one position, and confirms that the character at the beginning of the tape matches the character at the end. The read head then replaces this end character with an ‘o’, returns to the most recent ‘x’, and begins the process again, this time stopping its journey through the tape when it reaches the previous ‘o’. The algorithm has two ways it can reach the accept state, depending on whether the sequence is odd or even. If it is even, the process mentioned above will repeat until the read state is met with an ‘o’ - at this point, much like the parentheses problem, the TM has matched all the character pairs, and the accept state is reached. If the sequence is of odd length, the TM will read the middle character, replace it with an ‘x’, move right one position and read ‘o’, move back, and check the middle character again. Since it is an ‘x’, the algorithm can confirm that the character just read was the middle character, and moves to the accept state.

2.6 Additional Problems - Musical Alphabet

Recognise zero or more 3-digit binary strings, possibly separated by '#'s, and convert these strings into alphabetical characters according to the following schema:

000	a
001	b
010	c
011	d
100	e
101	f
110	g
111	-

Hashes are not required, but rather used to visually separate each individual string - a tape containing only hashes may be accepted, as they hold no intrinsic meaning. Given the tape #001#000#011#, the TM should recognise these strings and reach the accept state with the following tape: #xxx#xxx#xxx#bad. The tape 001000011 should also be accepted, resulting in the eventual tape xxxxxxxxxbad.



I found this problem quite fun to solve - if you didn't know, the letters a-g are used to represent notes in western octave-based music, hence the name Musical Alphabet.

On initialisation, the TM reads the first binary digit, replaces it with an x, and moves along the tape to the next binary digit, remembering what the previous digit was using the corresponding state. Another two digits are read, at which point the letter of the Musical alphabet has been determined - the

TM then travels along the tape until it reaches an empty character, replaces the empty character with the state's corresponding character, and then returns to the last 'x' it wrote, ready to repeat the process again. After the last sequence of binary digits has been translated, the TM returns to the read state, and travels up the tape; once it reaches an empty character on the tape before a binary digit, it can be confirmed that every character has been translated, and so the TM moves to the accept state.

3 Testing

3.1 StacsCheck Tests

REDACTED

3.2 Individual Tests

> Your program must be capable of dealing with a Turing machine with up to 10 000 states and 80 letters in the alphabet and simulating it for at least 100 million steps within the normal StacsCheck timeout of 10 CPU minutes.

I created a Turing Machine and corresponding tape to test my program's performance based upon the metrics mentioned above. This test, including the Python program used to generate all 100 million tape positions, can be found in the tests folder, along with a 'readme' file detailing how to run the test. To perform this test as accurately as possible, I disabled the printing of the tape, as it is considerably long and can cause StacsCheck to time out, despite my simulator having reached 100 million states before then. The following results were taken from a school computer, and prove that my Python implementation of the TM simulator reaches 100 million steps in 57 seconds, given a TM with 10,000 states and 80 letters. After confirming that my implementation met the minimum performance requirements, I wrote a variety of tests for my additional TMs that can be run using StacsCheck.

To ensure correct functionality of my palindrome TM, I wrote tests which included an empty tape, a regular incorrect tape, a tape that has an odd length, two long tapes - one odd, one even - a short tape, and a tape containing a single character. My implementation passes all these test cases. I then began testing my NAIVcipher, although due to the limited nature of the TM, the 'tests' were more of 'examples' of input and output. Nevertheless, the TM did produce the expected output, and passed all four tests. Finally, I wrote eight tests for my MusicalAlphabet TM - these tests included ordinary input, an empty tape, binary strings of invalid lengths, a tape containing just '#'s, and a long set of binary strings with no '#' separators. Once again, my TM implementation passed all eight tests.

All tests can be found in the 'Tests' directory, and can be run from the same directory using **REDACTED**, folder being whichever set of tests you wish to run. Do not attempt to run all tests at once, as the SpeedTest folder is contained within the same directory, which requires disabling of tape printing.

4 Complexity Analysis

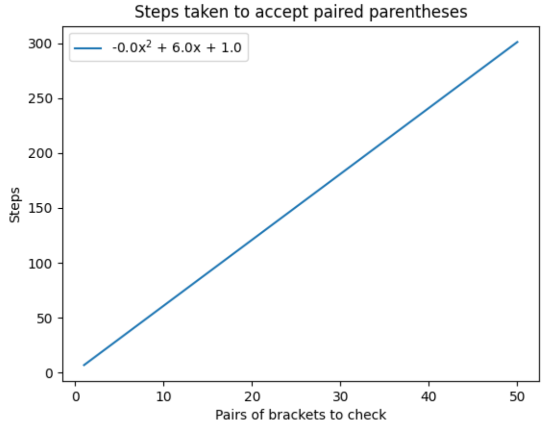
I will attempt to analyse the complexity of my algorithms both theoretically, and experimentally. Experimental analysis was performed on tapes using two different sample sizes: length 1 - 50, and length 1 - 100; this was to ensure trends were constant throughout, and to prevent against a trend which appears polynomial, but in reality is in the early stages of exponential time complexity. I left out the NAIVcipher TM's complexity from this report as it is obviously linear, but the graphs and data will remain in my submission should you want to see them.

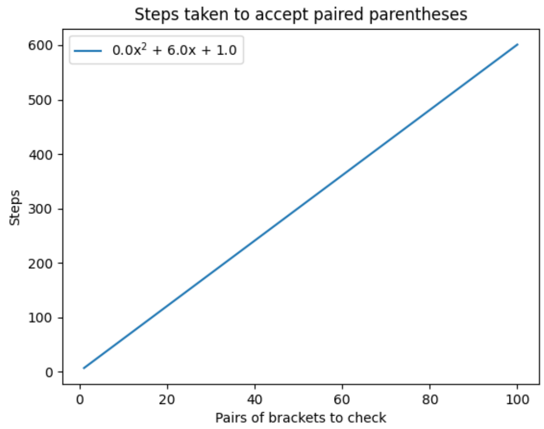
4.1 Specified Problems - Parentheses

I found analysing the complexity of this problem the most interesting, as there seem to be clear best case and worst case scenarios. Starting with the best case, a sequence of paired brackets - ()()()()()

The read head should move to the nearest 'close' bracket, replace it with an 'x', move left, replace the 'open' bracket with an 'o', and then move right twice; this process is then repeated 'n' times, n being the number of bracket pairs in the sequence. If the above process takes four steps, and it is being run n times, then the time complexity is approximately $4n$, which reduces to $O(n)$, linear.

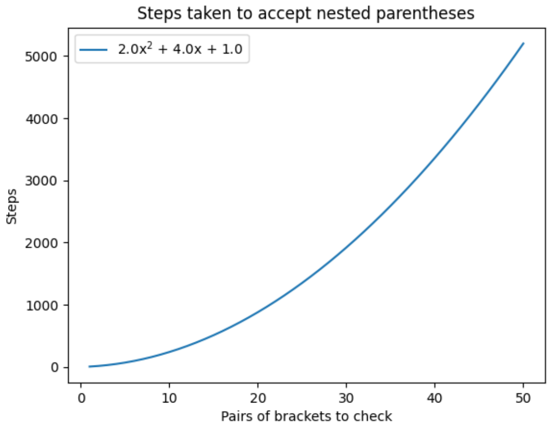
Analysing the algorithm experimentally, the theoretical prediction is confirmed, as $6n + 1$ reduces to $6n$, and then to $O(n)$ ignoring the constant.

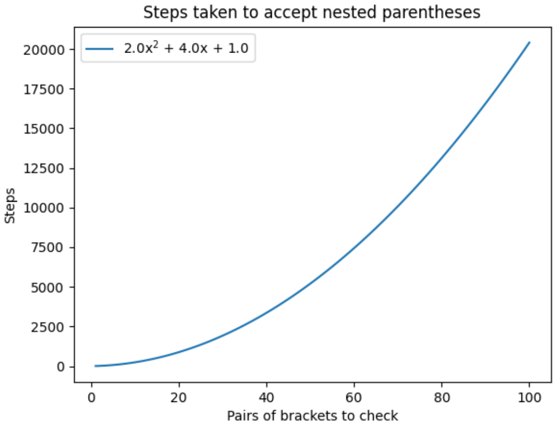




Addressing the worst case next, I believe that nested bracket pairs take the most steps to accept - for each bracket, the read head must move on average half the length of the tape, $\frac{n}{2}$. This must be done n times (n brackets), and so we should theoretically arrive at a polynomial time complexity, approximately $\frac{n^2}{2}$ which reduces to $O(n^2)$.

These predictions were confirmed experimentally, as $2n^2 + 4n + 1$ reduces to $O(n^2)$.

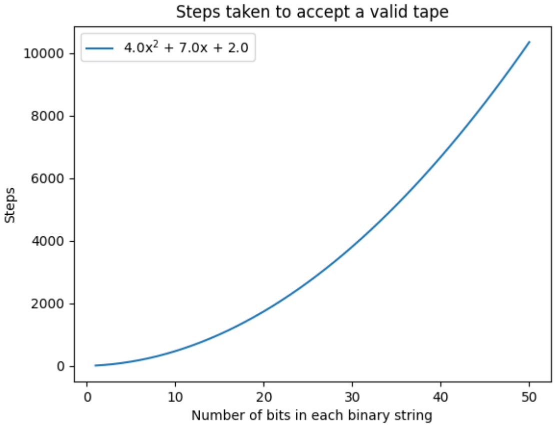


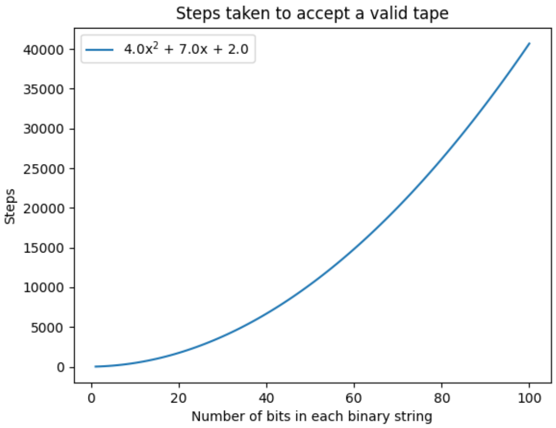


Therefore, while the parentheses problem has a best-case linear time complexity, the worst case is polynomial complexity.

4.2 Specified Problems - Binary Addition

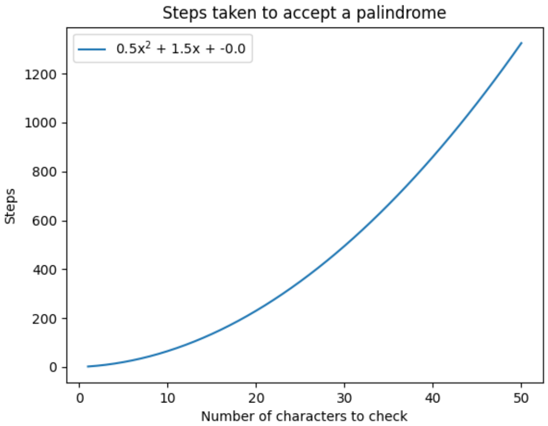
Processing a binary addition tape involves reading the first bit in each string - this involves travelling the majority of the tape. Discounting the '#'s between the strings, this operation is performed $\frac{n-2}{3}$ times, leading to a combined complexity of $\frac{(n-p)(n-2)}{3}$, where p is the number of positions not travelled when confirming each bit. This is quite clearly polynomial time complexity once again, and is backed up by the experimental evidence.

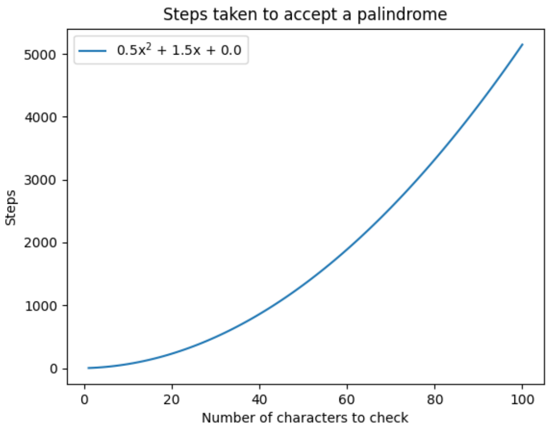




4.3 Additional Problems - Palindrome

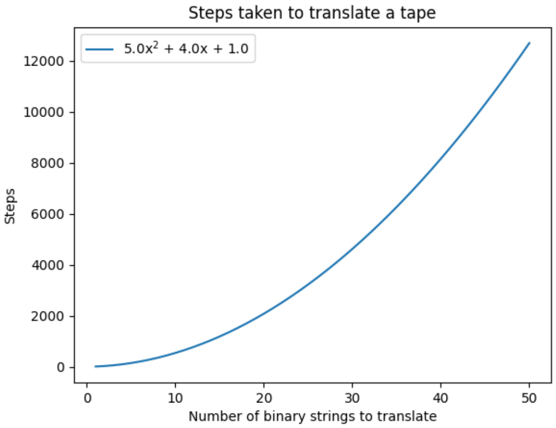
When checking a palindrome tape, the algorithm will check the first letter, then the last letter, then the second letter, and so on - the read tape moves across half of the tape on average, per move. We know there are n moves, as each letter must be checked - therefore the time complexity is approximately $n(\frac{n}{2})$, which reduces to $O(n^2)$. The experimental data once again corresponds with the theoretical analysis.

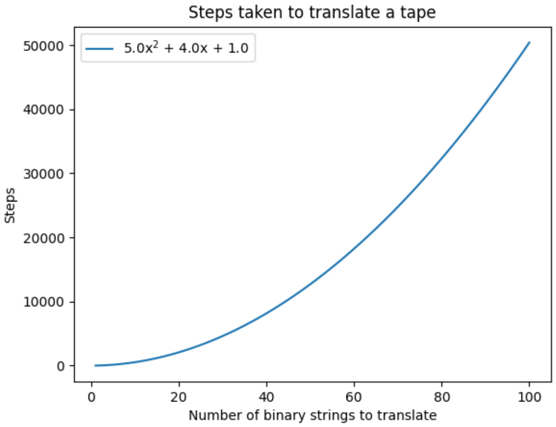




4.4 Additional Problems - Musical Alphabet

When processing a Musical alphabet tape, the TM reads a three-bit binary string, and then travels an average of $\frac{n}{2}$ steps; this is performed $\frac{n}{3}$ times, and so the resulting $\frac{n^2}{6}$ clearly reduces to $O(n^2)$, polynomial time complexity. Once again, the theoretical analysis is backed up by the experimental findings: $5n^2 + 4n + 1$ reduces to polynomial time.





5 Evaluation and Conclusion

I am satisfied with my TM simulator - I believe it satisfies all functionality requirements specified, and that the program's code is well commented, separated logically, and clearly demonstrates the flow of data throughout the simulation. I learnt about type hinting in Python while producing this piece of coursework, and so I made an effort to annotate every function with the appropriate data types, making the dataflow even clearer for someone who is unfamiliar with the program. Despite the success I have had using Python for this practical, I do recognise that my implementation is not the fastest it could be; in terms of performance, I would have been far better off using Java, or a compiled language. However, my program does meet the minimum performance requirements as shown in the Testing section, and using Python greatly improved my development experience, so I have no regrets.

My use of TM state diagrams in the design section should be very helpful as an aid to understanding how my TM solutions analyse their respective tapes. I am happy that I was able to provide solutions to the problems set in Task 2, but as mentioned in the Design section, I am now aware that I could have reduced the number of states in the binary addition solution, increased the alphabet, and kept the same functionality. This is an unfortunate oversight by myself, but it highlights the importance of reassessing completed work over time.

I feel the results of my complexity analysis are very conclusive - all but one of the algorithms turned out to have polynomial time complexity. I did find it interesting that upon analysis, I discovered that most, if not all of the algorithms involve extended travelling up and down the tape - I wonder, if given enough states and a sizable alphabet, could these algorithms be implemented in linear time, and how many states would be required.

I found it difficult to think of many tests that challenged unique functionalities of my additional TMs - given more time, I would like to design and implement more tests to ensure correct functionality and performance of my Turing Machines. Furthermore, I would be interested in implementing my TM simulator in Java, to see how drastic the improvement in performance would be. Having completed this practical, I now feel much more familiar with how Turing Machines operate, and what it takes to design an efficient algorithm.

References

[1] *Python Set remove() – Be on the Right Side of Change*. en-US. May 2021. URL: `https://blog.finxter.com/python-set-remove/` (visited on 06/17/2024).

[2] *TimeComplexity - Python Wiki*. URL: `https://wiki.python.org/moin/TimeComplexity` (visited on 06/17/2024).