CS3052 – Practical 1 (Turing Machines)

Introduction

For this practical I was required to design a program that would accept a textual description of a single-tape deterministic Turing Machine (TM), a file containing a number of inputs, and determine whether the machine provided would accept or reject the given inputs.

I was then asked to design several TMs of my own to solve a number of given problems, as well as several of my own, which will be described in due course. Following from this, I was asked to analyse how all of my solutions performed on various inputs as a function of the length of the inputs. This would help clarify their complexity, or – crudely – how well they scale with the problem sizes.

It is well known that a variety of problems can be solved in a certain amount of time. For example, determining whether a string is palindromic, with a modern computer and with access to memory, has time complexity of $O(n)$, meaning the time taken grows linearly with the size of the input. But a TM does not have access to memory, only state. This practical will assess how well this model of computing scales with the sizes of various problems.

Task 1

I designed a program `runtm` that would accept a TM description and internally represent it. Then it would go through a provided input file and determine whether each line of the file was accepted or rejected. I also decided from the outset to design an 'interactive' mode and a 'performance' mode using flags. The former would print out the transitions of the machine as it evaluated the input to assist debugging and verification, and the latter would print out the number of steps it took to complete the evaluation. An example of these flags in practice is shown:

```
Xaa_
<q0> <a> <q1> <X> <R>
Xaa_
<q1> <a> <q2> <a> <R>
Xaa_
<q2> <a> <q2> <a> <R>
Xaa_
<q2> <_> <q3> <_> <L>
XaX_
<q3> <a> <q4> <X> <L>
XXX_
<q4> <a> <q5> <X> <L>
XXX_
<q5> <X> <q6> <X> <R>
########################
Number of steps on input aaa_:
7
input accepted
```

Fig 1.0

To clarify, the input during each transition is displayed, followed by the transition it is currently in. The number of steps the machine takes to terminate is then displayed afterwards. The next line says whether the input was accepted or rejected.

N.B: I should note that this can get very lengthy, so future screenshots do not include the flags.

Then I decided to create test files based on the example in the specification (corrected after an error was discovered). These files would verify that the program was working as it was designed to do.

The TM was represented with an array of state objects, an arraylist of Characters to represent the input, an arraylist of transitions, and the current state/input. This meant that the code was fairly modular and split up naturally; meaning errors were easier to catch and debug.

There were a variety of test cases I came up with for the machine format, including, but not limited to:

- Testing for duplicate states or transitions
- Testing for an invalid state status
- Testing the sizes of both the alphabet and the set of states
- Testing for there being only one state per line
- Testing for invalid state names or invalid inputs/outputs
- Etc.

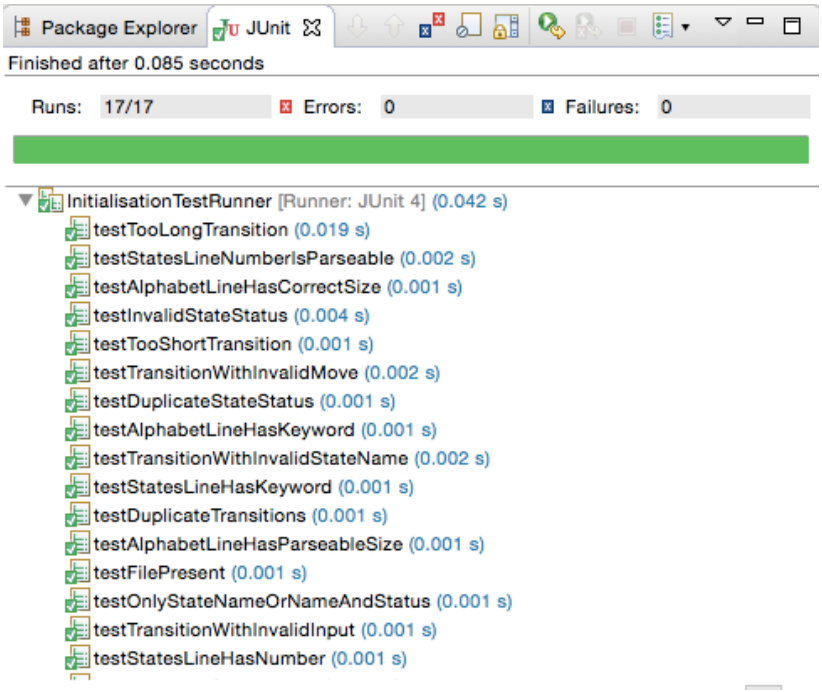

Fig 1.1

As Fig.1.1 shows, each of these tests were successful and the test cases/testing code is attached in the source code. The README.md file attached provides instructions for compiling and running the source code.

Task 2

The second task was to design TMs that would solve a number of problems – two provided, and three of my own choosing. For each of the solutions I would create two input files, one file full of inputs that were valid, and another of invalid inputs.

1. Recognise strings for the language of palindromes (strings that when reversed equal themselves) over the alphabet $\{a, b, c\}$. For instance, *aba*, *cabbbbac* and *aaaa* belong to the language, while *aab*, *ac* and *cabba* do not.

An obvious solution to this particular problem would be:

- Compare the first non-marked character to the last one
- Rewind to the next character to be compared
- Repeat until all characters are compared (i.e. at around the midway point)

However, I felt a simple enough improvement could easily be implemented which would roughly halve the time taken:

- Compare the first non-marked character to the last one
- Step back one
- Compare the current character to the first non-compared character
- Repeat until all characters are compared (i.e. at around the midway point)

The second algorithm was implemented by designing TMs that did just parts of the algorithm, and then I combined those TMs together to make a more complex TM. I should note my machine does check the input isn't marked on the first iteration, which substantially increases the number of states. This large number (26) is why I won't show the whole diagram, but the following diagrams are of some of its parts.

For clarification, the diagrams are of a palindromic TM for just the letter 'a'. The first diagram (Fig 1.2) handles moving forward, but reject if the input is already marked. The second diagram (Fig 1.3) deals with every occasion when the machine is going backwards. The third diagram (Fig 1.4) is similar to the first, but allows for marked input. I should note that the 'X' character is actually unnecessary, after discovering that the '_' character is sufficient for this purpose at the following link: [https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/four.html]

Once this machine was designed, I repeated the same process for the other letters, then conjoined the three machines together and factored out repeated states like the accept state. This process of building a complex machine out of simpler ones was highly effective and was a process I repeated with each of the subsequent solutions.

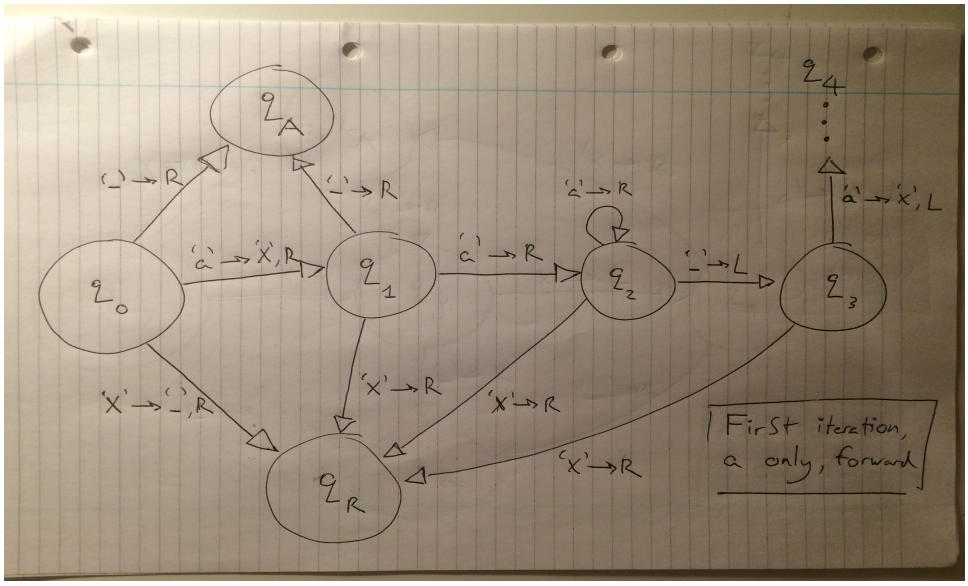

Fig 1.2

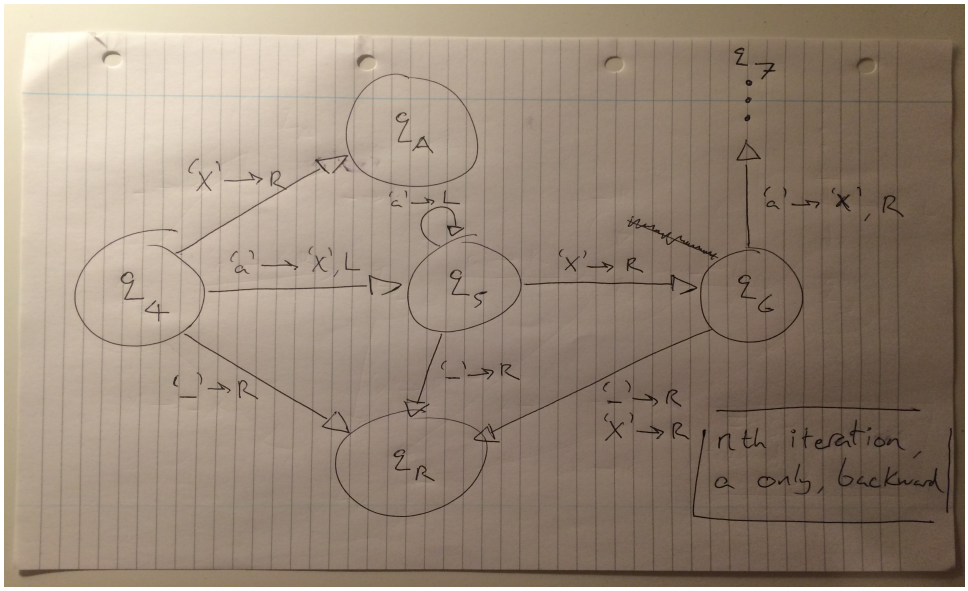

Fig 1.3

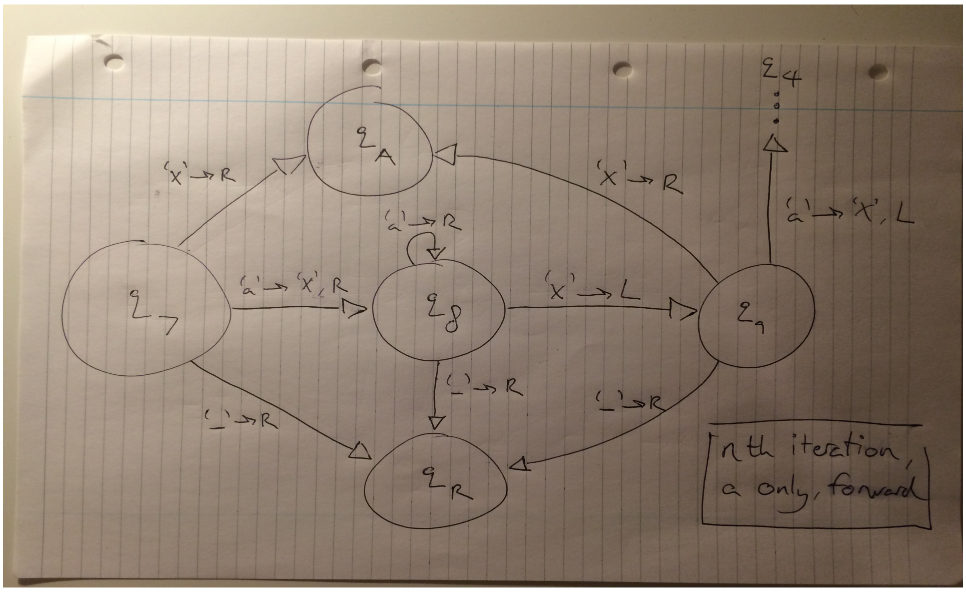

Fig 1.4

As you can see, none of these machines are particularly complex individually and combining them was a straightforward exercise. Hence I believe I have good reason to think the solution is correct. My tests strengthen this belief, as the test results below show:

```
8afbb6fd:java soutar$ java runtm machine_descriptions/palindrome.txt test_inputs/accept_palindrome.txt

input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
```

And for rejecting inputs:

```
Soutars-Air:java soutar$ java runtm machine_descriptions/palindrome.txt test_inputs/reject_palindrome.txt

input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
```

2. Recognise strings from the language {$w1\#w2\#w3$ | $w1$, $w2$, 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, and 0#0, 0#0#1, and 000#111#11 do not

The TM for this problem was implemented using a simple algorithm:

- Take the first number of the two arguments, mark them
- Check that their sum matches the first number of the result
- If it does, mark that number and rewind
- If they do match and the result produces a carry, mark that number and make the next comparison based on that assumption
- Repeat until all numbers are marked

This particular machine has 41(!) states, so I won't be showing the diagram. Nonetheless, all possible sums (including those with a carry) that can be produced in binary ([0,0], [0,1], [1,0], [1,1], [0,0,1], [0,1,1], [1,0,1], [1,1,1]) were accounted for, so it seems reasonable to suppose there is no chance of the algorithm being caught out. The tests also confirm that the machine accepts all correct test inputs:

```
Soutars-Air:java soutar$ java runtm machine_descriptions/binary.txt test_inputs/accept_binary.txt

input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
```

And rejecting all incorrect inputs:

```
Soutars-Air:java soutar$ java runtm machine_descriptions/binary.txt test_inputs/reject_binary.txt

input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
```

3. Recognise strings from the language {*w1#n* | w1 is an integer, and *n* is its digital root}. The digital root of a number is defined as the iterative sum of its digits, a process that repeats until the result is a single digit. For example, `digitalRoot(99)` => 9 + 9 = 18 => 1 + 8 = 9 . So 1234#1, 144#9, and 11111#5 belong to the language, and 100#100, 99, and 666#18 do not.

This machine turned out to be pretty straightforward. Because the digital root involves the modulo operator, there is no need to write out the sum of the digits and then iteratively whittle that number and its successors down. Rather, it can be found in one iteration of the input, with the final result being in the range of numbers 0 … 9, so I could simply store the sum in a single state. Consequently, the algorithm performs in linear time as opposed to polynomial time were it to write out the sum (since it would need to move backwards and forwards on each digit to increment the sum).

Like the others, there are tests to confirm the TM's correctness:

```
Soutars-Air:java soutar$ java runtm machine_descriptions/digital.txt test_inputs/accept_digital.txt

input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
```

```
Soutars-Air:java soutar$ java runtm machine_descriptions/digital.txt test_inputs/reject_digital.txt

input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
```

4. Recognise the language of identical strings over the alphabet {*a, b, c*}. For instance, abc#abc, abba#abba, and # (empty word = empty word) belong to the language, and abba#baab, abbaabba, and cac#ccc do not.

This is clearly a useful function in everyday computing (we compare strings quite often), so any algorithm would ideally be as efficient as possible. Much like the palindromic problem, a simple optimisation means the machine for this problem is twice as fast than the simplest machine possible, the algorithm being:

- Compare the first non-marked character of *w1* to the first non-marked character of *w2*
- Step forward one

- Compare the current character of *w2* to the first non-marked character in *w1*
- Repeat until all characters are compared (i.e. all the characters of both strings)

The machine diagram is very similar to that of the palindromic one – the only real difference being that there are states to check for the # character in the middle, and stepping forward instead of stepping backward. Given this similarity, implementing the machine was a swift process and the tests confirm that this machine is correctly implemented:

```
Soutars-Air:java soutar$ java runtm machine_descriptions/identity.txt test_inputs/accept_identity.txt

input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
```

```
^CSoutars-Air:java soutar$ java runtm machine_descriptions/identity.txt test_inputs/reject_identity.txt

input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
```

5. Recognise strings from the language {#*n*#*w* | *w* is an arbitrary sequence of characters and *n* is the length of *w*} over the alphabet {*a, b, c*}. For instance, #*6*#abcabc, #4#abba, and #0# belong to the language, and 6#abcabc, #4#abcba, and ## do not.

Given that we often use the lengths of strings in programming, a good algorithm would be ideal. However in this case that proved to be very difficult. A smarter computer scientist would have implemented the following, which I believe would have a time complexity of *O(n log n)*:

- Divide *n* by 2 (if odd, round down)
- Go through the input and mark off every other non-marked character (plus another if there was rounding down)
- Repeat until *n* is 0, and the input is completely marked.

However this was too tricky to implement correctly (dividing an odd number by 2 and 'cascading' numbers proved to be harder than supposed), and so instead I stuck with just decrementing *n* by 1 each time, and marking off a single character. Unfortunately this means the machine is not very fast and has a time complexity of $O(n^2)$. The good news was that this was the smallest machine in the number of states (20). Most of the states were just the ones charged with decrementing, since traversing the input was a pretty simple process. Again, the tests demonstrate the machine to be working as you would expect it to:

```
Soutars-Air:java soutar$ java runtm machine_descriptions/counter.txt test_inputs/accept_counter.txt

input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
input accepted
```

```
Soutars-Air:java soutar$ java runtm machine_descriptions/counter.txt test_inputs/reject_counter.txt

input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
input rejected
```

Task 3

The simplest way of testing how my machines perform was just providing specific test files as input to `runtm`. I should point out the lack of variety in the inputs is deliberate, because all bar one of the algorithms used are *stable*, meaning the characters in the input make virtually no difference to how long the relevant TM takes, only whether they are in the language and the length of the input.

This is obvious enough in the case of the palindrome and comparison TMs since comparing a 'b' with a 'b' is no more expensive than with two 'c's, but it is also true in the case of the digital TM (it just sums the digits and the result is always one digit); and finally the counter TM (it just blindly marks characters one at a time). The only slightly tricky exception is the binary TM, where the lengths of the two arguments may differ.

With that made clear, the following graphs show the number of steps taken by a given machine on an input, against the length of the input:

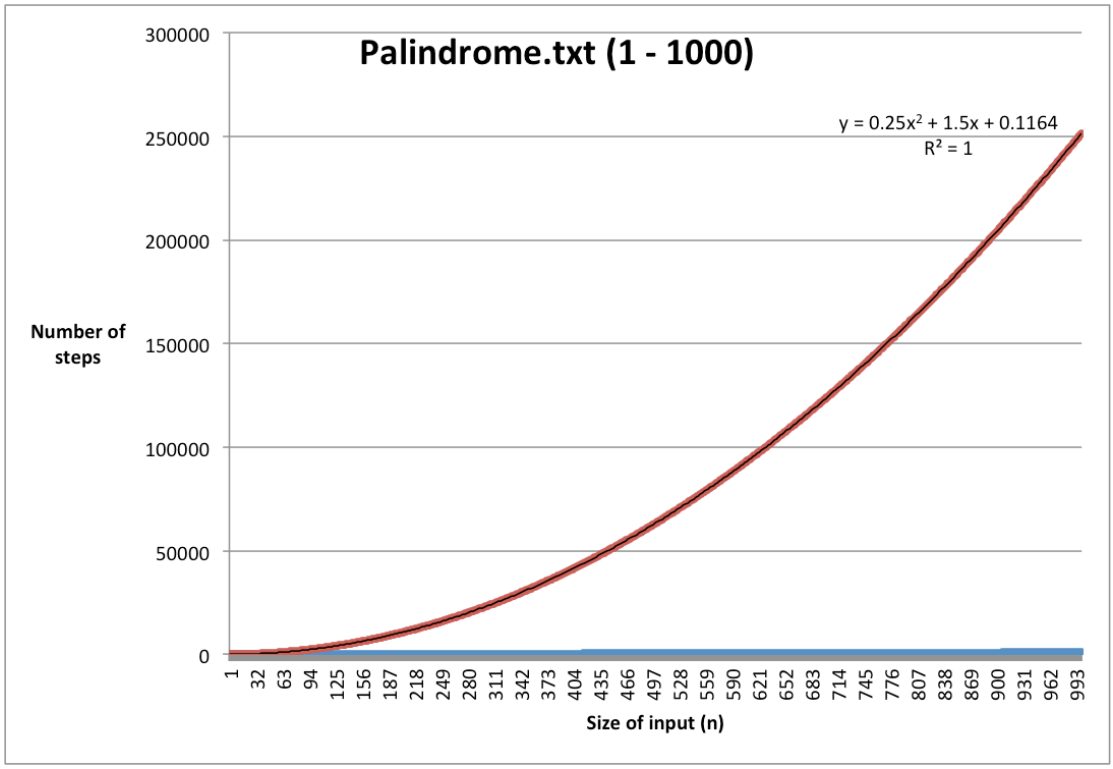

Fig 3.0

As Fig 3.0 shows, the TM represented by palindrome.txt has time complexity of $O(n^2)$ and more accurately $O(\frac{n^2}{4})$. This makes sense, as the machine oscillates and converges on the middle of the input, which is $\frac{n}{2}$, and it traverses across on average half of the input, which again is $\frac{n}{2}$. This results in $\frac{n^2}{4}$, which is in complete agreement with the data. The $1.5x$ is likely down to the fact that the machine must step back from a marked character on each iteration, and the 0.1164 simply being a constant to do with checking that the first character is not empty.

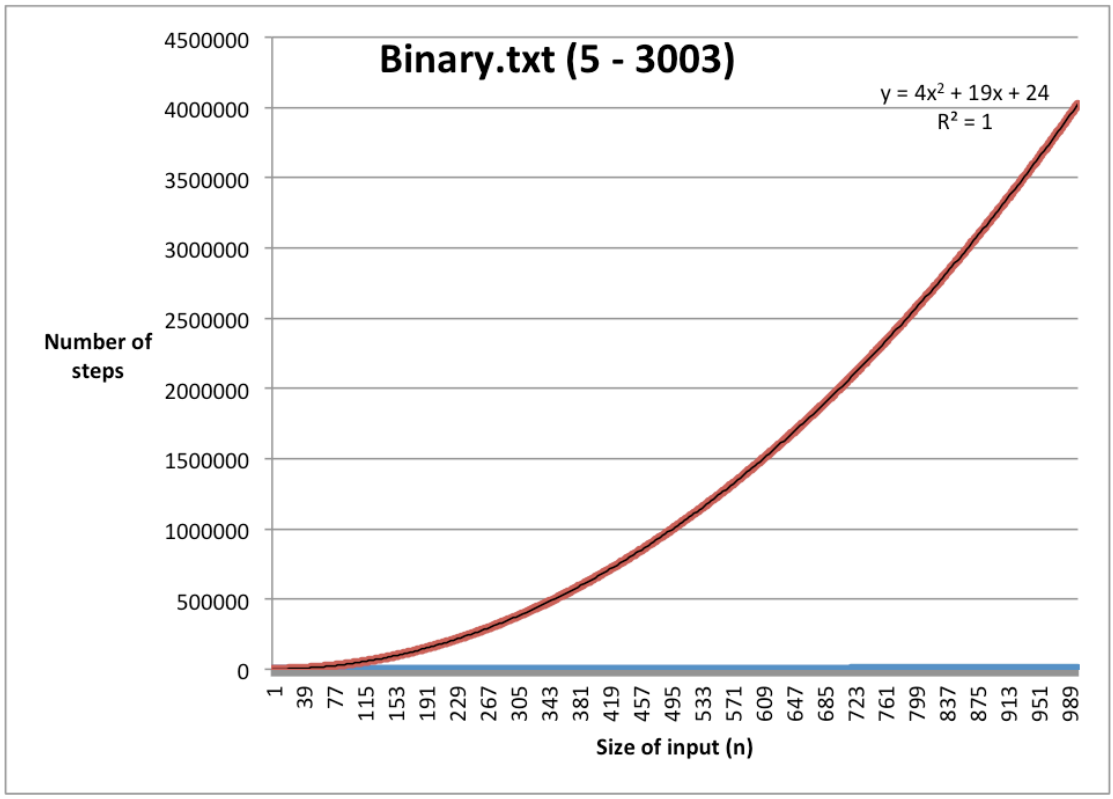

Fig 3.1

Fig 3.1 shows that the TM for binary.txt suffers from a time complexity of $O(n^2)$. The algorithm traverses over the length of both numbers and checks their sum, which is proportional to the larger of the two numbers. This is multiplied by the length of the two arguments, so if both are equal to each other than this is effectively $2n \cdot 2n$ (assuming n is the length of the summed numbers). Thus we have complexity of $O(4n^2)$, which is exactly what the data says. If the lengths of the two numbers to be summed are not equal, then the complexity is $O(mn)$ where m and n are the lengths of the two numbers respectively. $19x$ is likely to do with the length of the result (i.e. the third number).

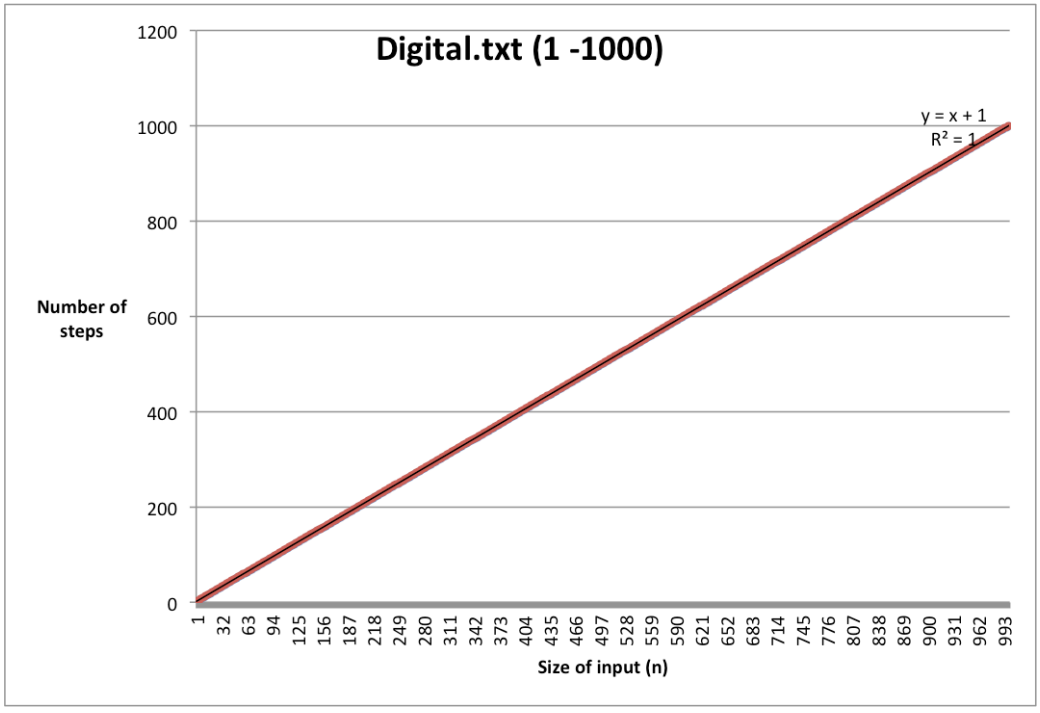

Fig 3.2

As Fig 3.2 shows, digital.txt has an impressive time complexity of *O(n)*. The sum is determined as it winds through the input in a single traversal, so there isn't even any need for marking characters. The constant of 1 would simply be reflecting the comparison of the result against the input.

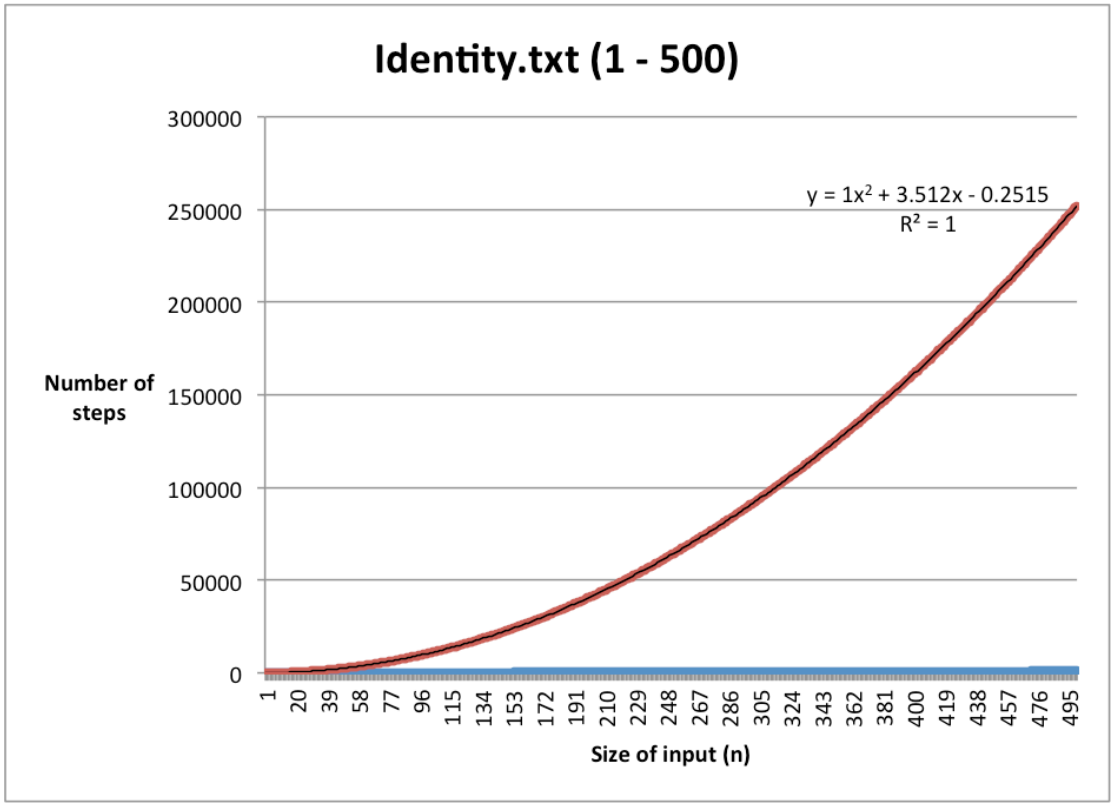

Fig 3.3

Unsurprisingly, given that two identical strings will be identical in length, the time complexity for identity.txt is $O(n^2)$, as Fig 3.3 shows. The reason why it is n^2 and not $2n^2$ is because it has to traverse the length of both strings, and when it traverses less of the first string it compensates in the second. But because the algorithm does comparisons in both directions, it cuts that number in half, giving just n^2.

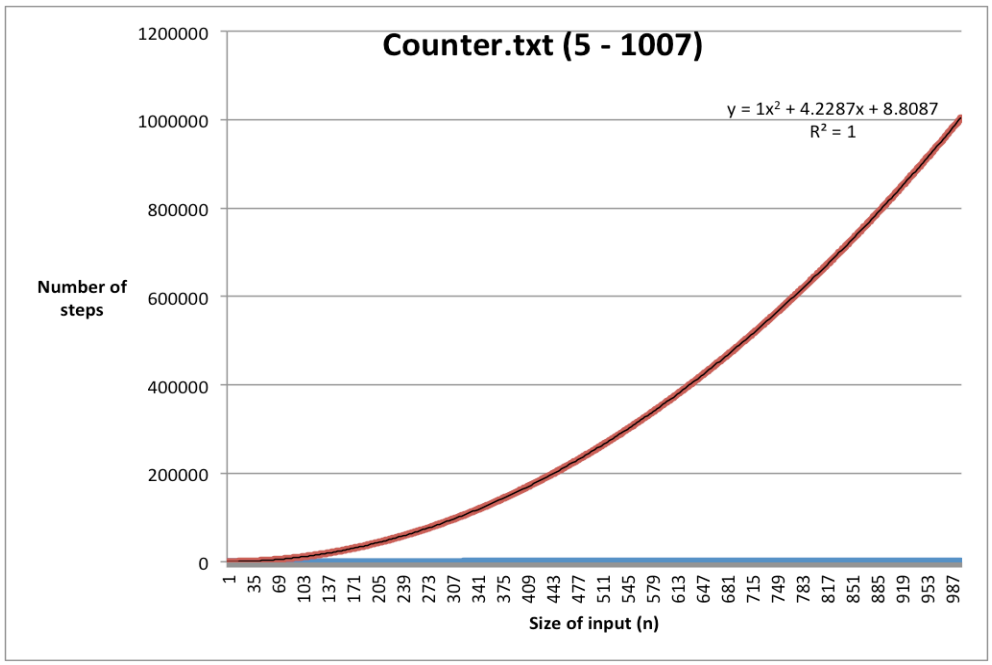

Fig 3.4

Finally, counter.txt is shown to have time complexity of $O(n^2)$ in Fig 3.4. This is unpleasant, if not surprising. The reason why is because it needs to decrement the value of n each time it marks a character, and because the value of n is meant to be equivalent to the length of the input itself, that will occur n times. On average the algorithm only traverses half of the input, $\frac{n}{2}$. But because it has to go back and forth with no work being done on the way back, that half is doubled, giving n, and combined with the number of iterations results in $O(n^2)$ time complexity.

Of greater value to us is how my TMs all perform on invalid input. And the short answer is quite well on many inputs – all of the machines that involve marking characters check to make sure there are no marked characters on the first iteration, so the length of time taken in such cases is linear. Hence I will not show graphs for those cases (they'd all be the same).

However, I will show graphs for other cases. In the case of palindrome.txt, the worst case is for the input to be palindromic right until a small difference at the very centre. This gives $O(n^2)$ performance and is virtually identical to the case where the input is palindromic, since as mentioned above my algorithms are mostly stable, including the one shown in Fig 3.5:

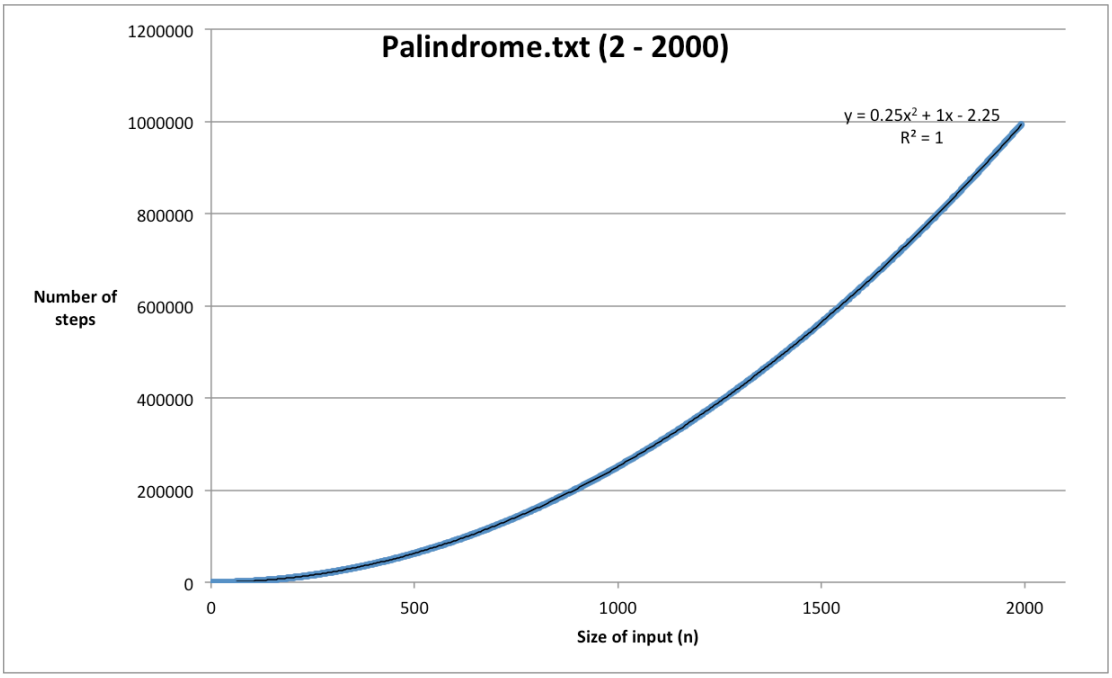

Fig 3.5

For binary.txt, the worst case regarding invalid inputs is when the numbers are both equal to each other in length and fail to match at their most significant bits. Any other case involving missing # characters or trailing 0s/incorrectly-summed most significant bits would be in *O(n)* time for the former and $O(n^2)$ for the latter. The graph for most-significant-bits-mismatch cases is shown in Fig 3.6:

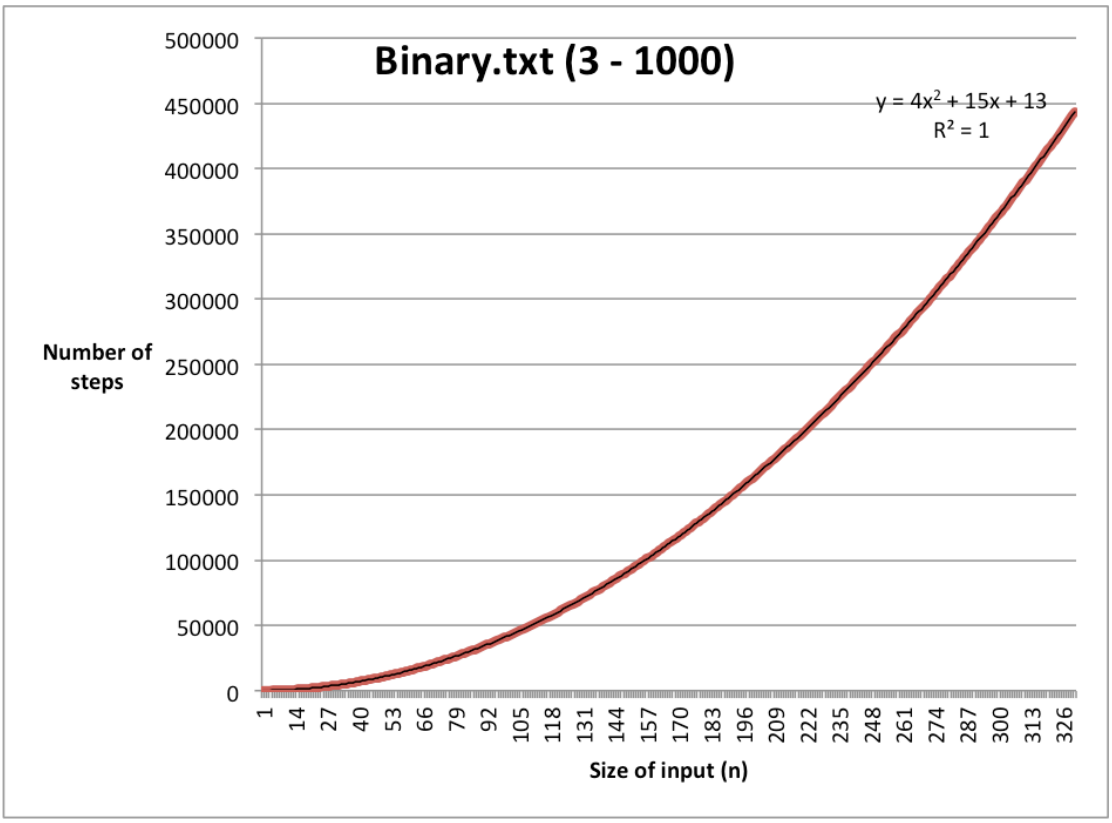

Fig 3.6

For digital.txt, the worst – and only – case where incorrect input doesn't involve characters outside the alphabet is when the digital sum gained by iterating through the input does not match the end result. This produces the exact same result as with any other case where the match is successful, as Fig 3.7 shows:

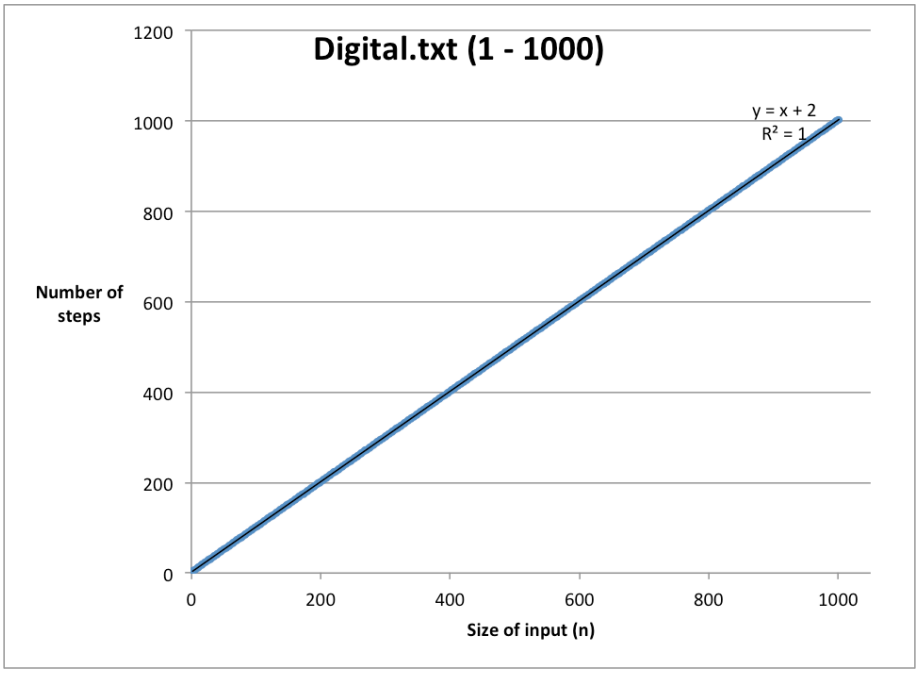

Fig 3.7

The worst case for identity.txt is when the strings are identical save for the last character. This will mean virtually identical time complexity of $O(n^2)$, with the graph of testing these cases shown in Fig 3.8:

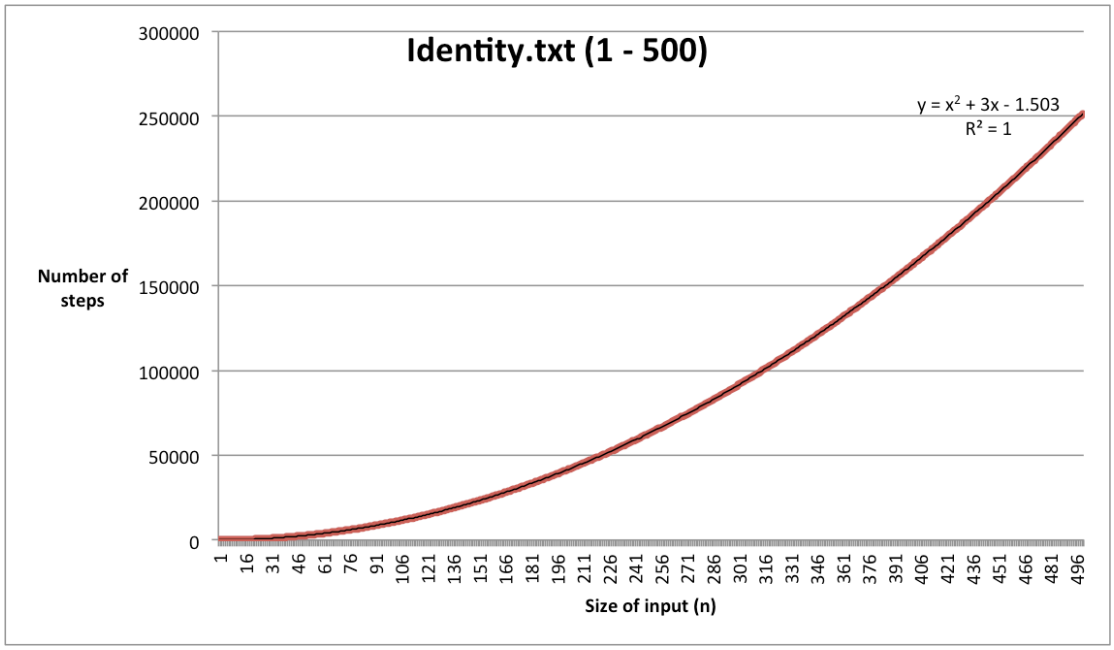

Fig 3.8

And finally, counter.txt has a worst case of $O(n^2)$ with invalid input since the input might be larger than the number provided. That said, it simply rejects as soon as it decrements n to 0 and subsequently finds an unmarked character, so it's no worse than if the input was correctly sized. The graph for these cases is identical and is shown in Fig 3.9:

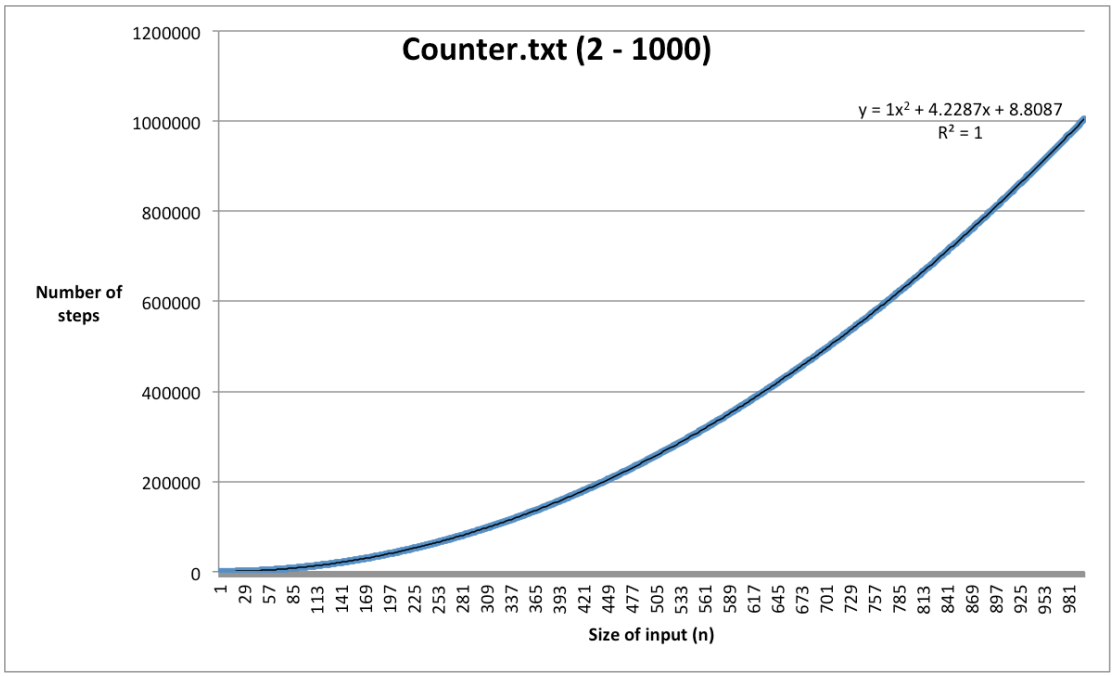

Fig 3.9

Conclusion

To conclude, this practical has clearly illustrated that the Turing Machine as a very simple model of computation is surprisingly powerful, as is clear from how it has handled several interesting – and in some cases, genuinely relevant – problems. Clearly in terms of performance it leaves much to be desired – for example, a multi-tape TM better describes how modern computers make use of memory and would result in all bar the digital machine being massively improved from $O(n^2)$ complexity to just $O(n)$ complexity.

Given more time, I discovered that the algorithm in palindrome.txt does not require the 'X' character since the '_' character is sufficient, so I could reduce the number of states used. The counter.txt algorithm could divide the number by 2 instead of decrementing it by 1 – this would result in time complexity of $O(n \log n)$ versus the $O(n^2)$ time complexity currently.

Without question, the most valuable thing gained from this practical was how dramatically my ability to both design and implement Turing Machines improved once I split up the processes into smaller sub-processes. The divide-and-conquer paradigm has never been more useful, particularly when the number of states or the size of the alphabet is large. It seems that complexity of design really can be reduced, and my observation has been that the simplest optimisations have given the best trade-off between the complexities of designing the machine versus the time complexity of the end product.