CS3052 Practical 1: Turing Machines

170011885

March 2019

Contents

1 Introduction

The purpose of this practical was the implementation of a Turing machine simulator, as well as the implementation of several Turing machines to solve different tasks and the analysis of their complexity.

2 Design of the Turing machine simulator

The simulator is mainly written in Python 3, but some optimizations are done in Cython. Apart from the main entry point, `src/runtm.py`, the Python code relevant to the simulators is located in the the `src/turing` directory which acts as a Python package.

Please note that the subsequent sub-sections will be brief and only highlight interesting aspects of the implementation. For more information, kindly refer to the commented code itself.

2.1 The `runtm` executable

The `Makefile` simply creates a symbolic link to `runtm.py`. This file has a shebang that directs the program loader to use the Python interpreter.

2.2 Parsing

Parsing is carried out in the `src/turing/parsing.py` file. Using customised exceptions, the program is able to generate quite exact error messages to direct the user to the source of the problem if there are syntax errors.

The parser uses the `TuringMachineDescriptionBuilder` to populate the states and alphabet (see next section).

2.3 Turing machine descriptions

The `TuringMachineDescription` class in `src/turing/description.py` holds all required information to define a Turing machine (formally the 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$). The transitions are encoded numerically for efficiency. This means that all states and all letters of the alphabet are assigned integer values and they are refereed to by these numbers. The reason for this is to improve efficiency. The `TuringMachineDescriptionBuilder` is in charge of encoding this information.

Also note that the `TuringMachineDescriptionBuilder` can be in a deterministic or nondeterministic mode. This affects the errors it generates, as well as how the transitions are stored. In any case, the transitions are stored in a Python `dict` (which is essentially a hashmap).

2.4 The deterministic Turing machine simulator

The Turing machine simulators are located in the `src/turing/machine.py` file. The deterministic simulator is represented by the `DeterministicTuringMachine` class. The `process_input` method takes as argument the contents of the tape and repeatedly calls the `perform_step` method that is responsible for performing one transition (including writing to the tape). Each iteration modifies the instance of `DeterministicTuringMachineConfiguration` in place which represents the configuratino of the TM.

Upon reaching a halting state, the `process_input` returns an instance of `TuringMachineResult` that contains the tape content, number of transitions, and whether the machine accepted or rejected.

The `process_input` method also has a verbose option which is triggered when running `./runtm -v ...`. When this is enabled, it will print the contents of the tape after each step, along with the name of the current state. This requires doing reverse lookups for the tape and states at each step, so it should only be used for debugging.

2.5 The nondeterministic Turing machine simulator

The `NondeterministicTuringMachine` class works similarly to its deterministic counterpart, except that at each step, it looks at all possible configurations the Turing machine could be in. Note that the configurations are stored in a `numpy` array. This is to allow easier interfacing with the `Cython` code. This is more clearly documented in the comments of the `src/turing/optimisations.pyx` file which is in charge of handling the 'bottlenecks' that were encountered when testing the nondeterministic machines. This file is compiled to `C` for efficiency.

3 Deterministic Turing machines

This section will examine the deterministic Turing machines that were implemented.

3.1 The Turing machine M_{paren}

This Turing machine should recognize sequences of balanced parentheses.

3.1.1 Description

High-level description Repeatedly, we move right to the next), replace it with X, then we move left until we find the next (and replace it with X.

If when we are looking for a) and we reach the end of the tape (␣), we accept if the tape only contains X. Note that we need to introduce a special start of tape symbol and take care of the ramifications of introducing this new symbol.

Implementation-level description M_{paren} = "on input $w \in \{(,)\}^*$:

1. If the initial cell is blank, accept; if it is), reject. Otherwise, replace the (with Y.
2. Sweep right until the next) and replace it with X.
   - If in sweeping right we encounter ␣, scan the tape left until \$ and reject if we encounter any (,) or Y. If, however, we do reach \$, accept.
3. Sweep left until the next (and replace it with X. Or, if we reach Y, replace it with \$. In either case, repeat from step 2.
   - If in sweeping left we encounter \$, reject."

Notice that the tape alphabet $\Gamma = \{(,), X, Y, \$\}$; it introduces three new letters.

3.1.2 Theoretical analysis of complexity

For all $n \geq 0$, there exists a particular input word $x \in \{(,)\}^n$ that maximizes the number of steps that M_{paren} must perform until it halts, i.e. all other input words of that length either need the same number of steps or fewer.

This worst-case scenario arises when M_{paren} must:

- sweep the longest distances because this causes more transitions; and
- perform the most sweeps because the input word cannot be rejected (or must be rejected as late as possible.

We find that the worst possible input word is a sequence of strictly nested parentheses because this input word satisfies both conditions. Formally, these words are $x_n = (^{\lceil n/2 \rceil})^{\lfloor n/2 \rfloor}$, i.e. open parentheses followed by the same number of close parentheses, prepended by an additional open parenthesis if the length of x should be odd.

We will now attempt to define the function $f(n)$ that returns the maximum number of transitions made by the TM for any word of length n.

Complexity for even length words For even n:

- we reach the first) after moving right to the middle of the word ($n/2$ steps);
- we then move left $i_1 = 1$ time, right $i_2 = 2$ times, left $i_3 = 3$ times, ..., until we move right $i_n = n$ times (totalling $\sum_{i=1}^{n} i$ steps);
- at that point we reach _, so we need to move left to the beginning of the word to verify that we have crossed of all parenthesis (n steps);
- arriving at the left end of the tape ($), we accept the word.

Counting the steps, we obtain

$$f_{\text{even}}(n) = \frac{n}{2} + \sum_{i=1}^{n} i + n.$$

Using the formula for sums of arithmetic sequences, we simplify to

$$\begin{aligned} f_{\text{even}}(n) &= \frac{n}{2} + \frac{n}{2}(1+n) + n \\ &= \frac{n^2}{2} + 2n. \end{aligned}$$

This can be visually verified by running `./runtm -v` on the `paren.tm` machine with an apprpriate input word. The `-v` option will print the tape at each state along with the position of the read head marked in red. The screenshot on the right shows the operation on the input `((()))`.

Complexity for odd length words For input words of the form $x_n = (^{\lceil n/2 \rceil})^{\lfloor n/2 \rfloor}$:

- we find the first) after $(n+1)/2$ steps;
- then we follow the same procedure (move left and right from $i_1 = 1$ to $i_n = n - 1$) (totalling $\sum_{i=1}^{n-1} i$ steps);
- at that point we reach _ and need to move left to the beginning of the word (n steps), concluding that that the word should be rejected.

We obtain

$$\begin{aligned} f_{\text{odd}}(n) &= \frac{n+1}{2} + \sum_{i=1}^{n-1} i + n \\ &= \frac{n+1}{2} + \frac{n-1}{2}(1+n-1) + n \\ &= \frac{n^2}{2} + n + \frac{1}{2}. \end{aligned}$$

Complexity class For all integers $n > 1$, $f_{\text{even}}(n) > f_{\text{odd}}(n)$ because

$$\frac{n^2}{2} + 2n > \frac{n^2}{2} + n + \frac{1}{2}.$$

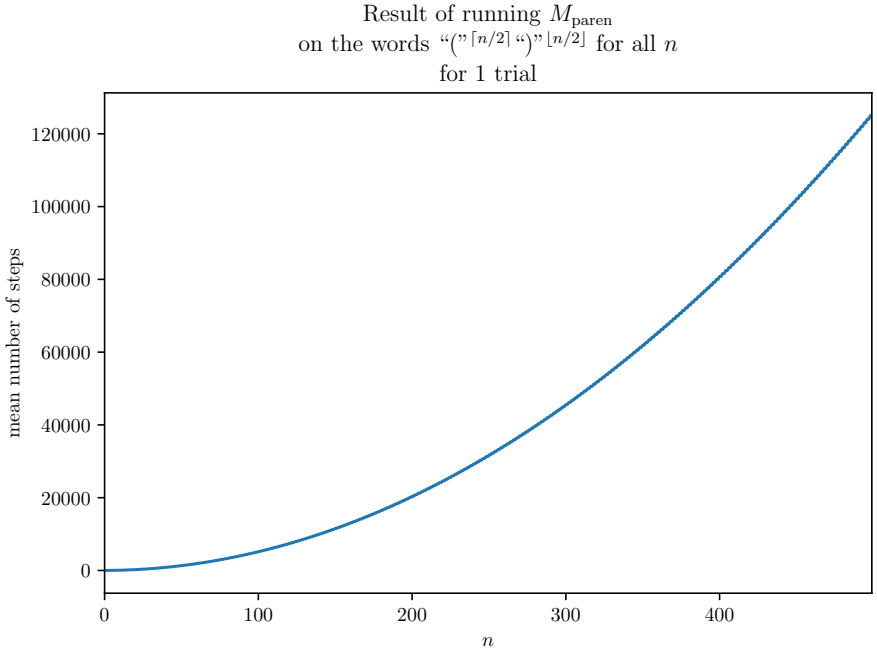
That only leaves us to consider $n = 0$, but in this case f_{even} applies, so for practical purposes $f_{\text{even}}(n) > f_{\text{odd}}(n)\ \forall n$, so we can conclude that M_{paren} is in the complexity class

$$\text{DTIME}\left(\frac{n^2}{2} + 2n\right).$$

Therefore the TM has $\mathcal{O}(n^2)$ complexity.

3.1.3 Practical analysis of complexity

The theoretical results are evident in practical trials. The data collected in `data/paren-nested.csv` confirms the findings of f_{odd} and f_{even} (the values were cross-checked). Furthermore, the plot below shows how this worst-case scenario behaves as a plot of n against the number of transitions made by the machine.



Other types of input words were also tested; please refer to the plots in section 8.1, as well as the CSV data in `data/paren-*.csv`.

3.2 The Turing machine M_{binadd}

This Turing machine should recognize the language

$$L_{\text{binadd}} = \{w_1\#w_2\#w_3 \mid w_1, w_2 \text{ and } w_3 \text{ are binary numbers, least significant bit first, and } w_1 + w_2 = w_3\}.$$

3.2.1 Description

High-level description Move right along the tape, remembering the first 0 or 1 in the first two words, and replacing them with X. Then, compare the sum of these two symbols with the first number of the third number. Move back to the start and repeat this procedure until the first two numbers consist only of Xs. If the third only consists of Xs followed by 0s (i.e. the tape looks like $X^*\#X^*\#X^*0^*$), accept.

If the fist two numbers are of different lengths, the shorter one is (for the purpose of the calculations) padded with 0s after the the number itself. These 0s should not be written to tape, but simply imagined. Refer to the implementation-level description for more information.

Implementation-level description M_{binadd} = "on input $w \in \{0, 1, \#\}^*$:

1. Remember[1] the value $c = 0$. This will represent the carry bit.
2. Read the current cell and remember its value (0, 1, or #). We will call it u.
3. If $u \neq \#$, write X to the tape and sweep right until the next #. This marks the end of the first word.
4. Sweep right until the current cell does not read X. Remember the contents of that cell (we will call it v).
5. If $v \neq \#$, write X to the tape and sweep right until the next #. This marks the end of the second word.
6. Sweep right until the current cell does not read X.
   - If $u = v = \#$ and $c = 0$, read until the end of the tape ($\_$). Accept if we encounter only 0s until we reach the end of tape symbol. Reject otherwise.
   - If $u = v = \#$ and $c = 1$, read the current cell. Reject if it is not 1. Accept if it is $\_$. Otherwise, read until the end of the tape ($\_$). Accept if we encounter only 0s until we reach the end of tape symbol. Reject otherwise.

[1] The term "remember" refers to creating as many states as necessary such that the different cell values follow different paths, but essentially, a specific path represents a specific value that was encountered.

- Otherwise (i.e. if $u \neq \#$ or $v \neq \#$) read the value of the current cell, remember it as t, and replace its contents with X. To continue, consider the following table.

current value				remember
c	u	v	t	c
0	$0, \#$	$0, \#$	0	0
0	$0, \#$	1	1	0
0	1	$0, \#$	1	0
0	1	1	0	1
1	$0, \#$	$0, \#$	1	0
1	$0, \#$	1	0	1
1	1	$0, \#$	0	1
1	1	1	1	1

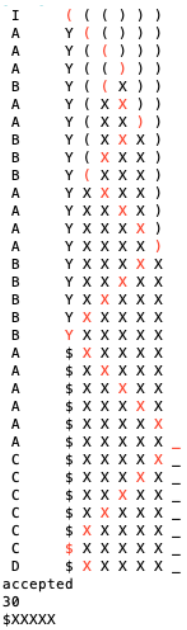
  If the current set of values of c, u, v, t is not represented in the table, reject. If it is, set c to the respective value of the remember column and move on to the next step.
- Move left until we encounter #. This marks the end of the second word. Move left until we encounter # again. This marks the end of the first word.
- Move left until we encounter X. Then move right one cell and go to step 2."

3.2.2 Theoretical analysis of complexity

Let $f(n)$ denote the maximum number of steps for an input word of length n again. The word that maximizes the number of steps required to reach a halting state for $n > 2$ is $x_n = \#0^{n-2}\#$. This is because the machine will have to cross the entirety of the middle word each time, marking off one letter at a time, until reaching the accepting state.

For these input words:

1. We move right for n steps to reach the end of the tape. While doing so, we marked off the next unmarked 0 of the middle word.
2. We move left to the beginning of the tape (n steps).
3. We require two more transitions to return to the initial step (2 steps).
4. We repeat steps 1-3 for a total of $n-2$ times because that is the length of the middle word (it will be marked off entirely).
5. Finally, we need to move to the end of the tape to make sure there were no letters left in the final word (n steps).



The screenshot on the right shows an example for the input word #000#.

Steps 1 to 3 require a total of $2n+2$ transitions. These steps are carried out a total of $n-2$ times, so we have $(n-2)(2n+2)$ transitions. Finally, the last step requires n transitions. We arrive at

$$\begin{aligned} f_{n>2}(n) &= (n-2)(2n+2)+n \\ &= 2n^2-n-4. \end{aligned}$$

Brute forcing the possible input words of length $0 \leq n \leq 2$, we find $f(0)=0$, $f(1)=1$, and $f(2)=2$. To obtain $f(n)$ for all n, we simply take $f_{n>2}+4$, yielding

$$f(n) = 2n^2-n.$$

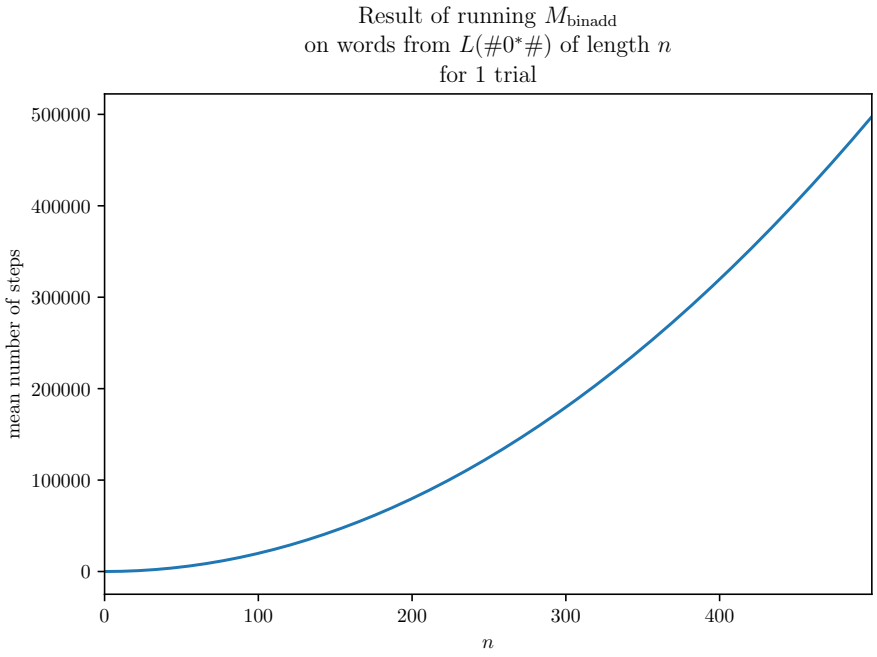
As a result,

$$M_{\text{binadd}} \in \text{DTIME}(2n^2-n),$$

and so its complexity is in the order of $\mathcal{O}(n^2)$.

3.2.3 Paractical analysis of complexity

The theoretical results are evident in practical trials. The data collected in `data/binadd-accepting_zeros.csv` confirms the findings of f (the values were cross-checked). Furthermore, the plot below shows how this worst-case scenario behaves as a plot of n against the number of transitions made by the machine.



Other types of input words were also tested; please refer to the plots in section 8.2, as well as the CSV data in `data/binadd-*.csv`.

3.3 The Turing machine $M_{\textbf{binaryunary}}$

This Turing machine, on binary input $w \in \{0,1\}^*$ (least significant bit first) should halt in the accepting state leaving the unary equivalent of this number on the tape. On other words, if the input w represents the number $x \in \mathbb{N}$, the contents of the tape upon halting in the accepting state should be 1^x.

The table below shows some examples.

tape input, w	tape output
␣	␣
1	1
10	1
01	11
011	111111

3.3.1 Description

High-level description Repeatedly, we decrement the binary number by one and append a # to the end of the tape. Once the binary number is 0, move each # to the beginning of the tape, converting each to a 1.

Implementation-level description $M_{\text{binaryunary}}$ = "on input $w \in \{0,1\}^*$:

1. Move every tape symbol one to the right and make the leftmost tape symbol to be \$. While this is not strictly necessary (and will increase the number of transitions required), this will make the implementation easier.
2. Move right until we encounter a 1.
   - If, in doing so, we encounter a ␣, instead clear the tape (replace all cells with ␣) and accept.
   - If, in doing so, we encounter a #, instead move one cell left and go to step 7.
3. Write Y to that cell and move right until we encounter ␣.
4. Write # to that cell and move left until we encounter Y.
5. Write 0 to that cell and move left until we reach \$. In doing so, replace each 1 with 0, but leave the 0s as 0s.
6. Go to step 2.
7. Move left (reading 0s) until either of the following conditions holds:
   - we encounter a 1, in which case we move one right and write 1 to that cell (on the right); or
   - we encounter a \$, in which case we write 1 to the tape and move one cell to the right.

8. Move right while reading 0s, until:

   - we encounter a #, in which case we write 0 to the tape, move one left and go to step 7; or
   - we encounter the end of the tape (_), in which case we move left while reading 0s and replacing them with 1, and once we encounter 1, accept."

3.3.2 Theoretical analysis of complexity

Clearly, the word x_n that will require the most transitions for a particular n will be of the form $x_n = 1^n$, as this will be the largest binary number that can be represented in that space, and as a result, the machine will need to write the most 1s to the tape.

On input $x_n = 1^n$ (as seen in the example screenshot to the right where $n = 2$):

1. We move to the end of the tape (n transitions). Then we move each symbol one to the right, and set the leftmost symbol to \$ ($3n$ transitions).

2. We move over the binary number, marking the first 1 we encounter (n transitions). If we do not encounter a 1 until reaching # or _, go to step 5.

3. Let i denote the current number of # characters at the end of the word (initially $i = 0$). We need to move past all the # characters and append an extra # at the end ($i + 1$ transitions).

4. We move back to the start of the tape, decrementing the binary number by one while doing so ($i + n + 1$ transitions). Then, we go to step 2 (this means steps 2-4 will be repeated until $i = 2^n - 2$).

5. Write 0 to the current cell and move to the beginning of the tape ($n + 1$ transitions).

6. Write 1 into the current cell and move right until encountering # ($n + 2$ steps).

7. Write 0 into the current cell and move left until encountering 1 ($n + 2$ steps).

8. Write 1 into the current cell and move right until encountering # ($n + 3$ steps). Go to step 7 (total of $2^n - 2$ repeats).

- If, instead, we encounter ␣, move left, writing ␣ to the tape while reading # and accept ($n + 2$ steps).

Counting the number of transitions in terms of n, we obtain

$$\begin{aligned}
f(n) &= n + 3n + \sum_{i=0}^{2^n-2} (n + i + 1 + i + n + 1) + 2(n + 1) + n + 2 \\
&\quad + (n + 2 + n + 3)(2^n - 2) + n + 2 \\
&= (2n + 5)(2^n - 2) + 8n + 6 + \sum_{i=0}^{2^n-2} (2n + 2i + 2) \\
&= (2n + 5)2^n + 4n - 4 + \sum_{i=0}^{2^n-2} (2n + 2i + 2) \\
&= (2n + 5)2^n + 4n - 4 + \frac{2^n - 1}{2}(2n + 2 + 2n + 2(2^n - 2) + 2) \\
&= (2n + 5)2^n + 4n - 4 + \frac{2^n - 1}{2}(4n + 2(2^n)) \\
&= (2n + 5)2^n + 4n - 4 + (2^n - 1)(2n + 2^n) \\
&= (2^n + 4n + 4)2^n + 2n - 4 \\
&= 4^n + n2^{n+2} + 2^{n+2} + 2n - 4.
\end{aligned}$$

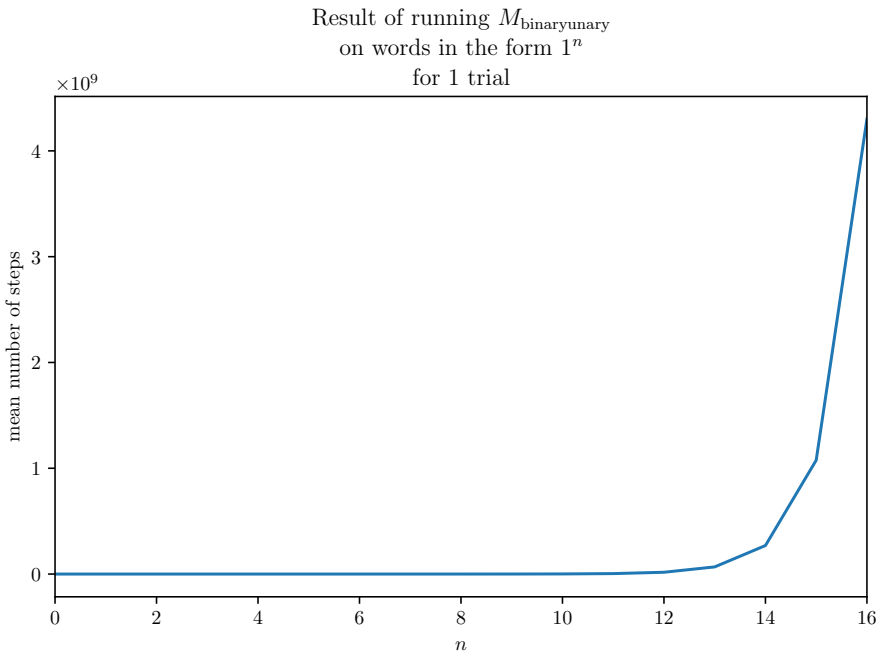
As a result,

$$M_{\text{binaryunary}} \in \text{DTIME}(4^n + n2^{n+2} + 2^{n+2} + 2n - 4),$$

and so its complexity is in the order of $\mathcal{O}(4^n)$.

3.3.3 Paractical analysis of complexity

The theoretical results are evident in practical trials. The data collected in `data/binaryunary-ones.csv` confirms the findings of f (the values were cross-checked). Furthermore, the plot below shows how this worst-case scenario behaves as a plot of n against the number of transitions made by the machine. Note that the number of steps explodes so quickly that we only plot until $n = 16$.



Other types of input words were also tested; please refer to the plots in section 8.3, as well as the CSV data in `data/binaryunary-*.csv`.

3.4 The Turing machine M_{subword}

This Turing machine should recognize the language

$$L_{\text{subword}} = \{w_1 \# w_2 \mid w_1 \in \{0,1\}^* \text{ and } w_2 \text{ is a substring of } w_1\}\,.$$

3.4.1 Description

High-level description Repeatedly, we compare the first word to the second. If the first word starts with the second, accept. Otherwise, mark off the first unmarked letter from the first word and repeat. If the first word is completely marked off, reject.

Implementation-level description M_{subword} = "on input $w \in \{0,1,\#\}^*$:

1. Read the current cell, remember its value as c, and write \$ to it.
   If $c = \#$, move right one cell and accept if that cell's value is $\_$, reject otherwise.

2. Move right until #, then move right while reading Xs and Ys.

3. If the current cell has the same value as c, write X if $C = 2$ or Y if $c = 1$ to the current cell and move left until encountering either \$, X, or Y.
   Otherwise, if the current cell's value is _, accept.
   Otherwise, go to step 5.

4. Move right one cell. Read this cell's value, remember it as c, and write Y to it. Go to step 2.

5. Move left until reading \$. While moving left, replace all X with 0 and Y with 1.

6. Move right one cell and go to step 1."

3.4.2 Theoretical analysis of complexity

The word x_n that will require the highest number of transitions for a particular $n > 2$ is of the form $x_n = 1^{n-3}\#10$. Note that the word $0^{n-3}\#01$ would be equivalent in respect to the fact that it will require the same number of steps, but we will look at x_n.

On input $x_n = 1^{n-3}\#10$ for $n > 2$ (as seen in the example screenshot to the right where $n = 3$):

1. Let i denote the number of letters marked off the first word. Initially, $i = 1$. Mark off the next letter of the first word and move to the beginning of the second word ($n - i - 1$ transitions).

2. Mark off the first letter of the second word and move left until reading the marked character of the first word ($n - i - 1$ transitions).

3. Read the cell to the right and compare that to the second letter of the second word ($n - i$ transitions).

4. Move back to the beginning (because the letters did not match) ($n - i$ transitions)

5. Move one cell to the right (1 transition). If $i \leq n - 1$, go to step 1.

6. Now only 7 transitions remain, after which the word is rejected.

Counting the number of transitions we obtain

$$\begin{aligned} f(n) &= \sum_{i=1}^{n-1}(n-i-1+n-i-1+n-i+n-i+1)+7 \\ &= \sum_{i=1}^{n-1}(4n-4i-1)+7 \\ &= \frac{n-1}{2}(4n-4-1+4n-4(n-1)-1)+7 \\ &= \frac{n-1}{2}(4n-2)+7 \\ &= (n-1)(2n-1)+7 \\ &= 2n^2-3n+6 \end{aligned}$$

for $n > 2$.

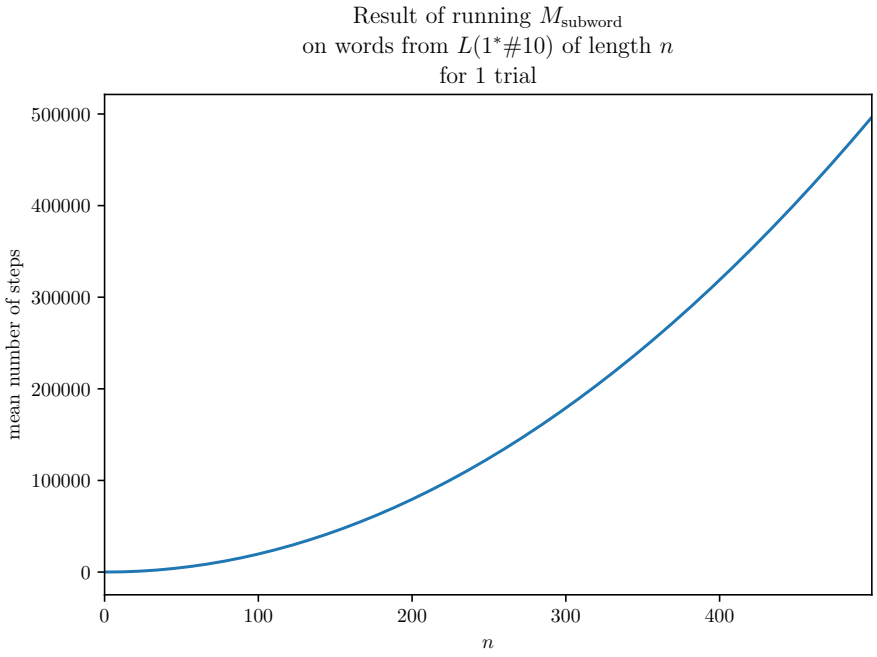
Manual testing showed that $f(0) = 0$, $f(1) = 1$, and $f(2) = 2$, so we can use $f(n)$ for all n and be convinced that for any n, the value of $f(n)$ will be greater or equal to the number of transitions required. Hence

$$M_{\text{subword}} \in \text{DTIME}(2n^2-3n+6),$$

and so its complexity is in the order of $\mathcal{O}(n^2)$.

3.4.3 Practical analysis of complexity

The theoretical results are evident in practical trials. The data collected in `data/subword-ones.csv` confirms the findings of f (the values were cross-checked). Furthermore, the plot below shows how this worst-case scenario behaves as a plot of n against the number of transitions made by the machine.

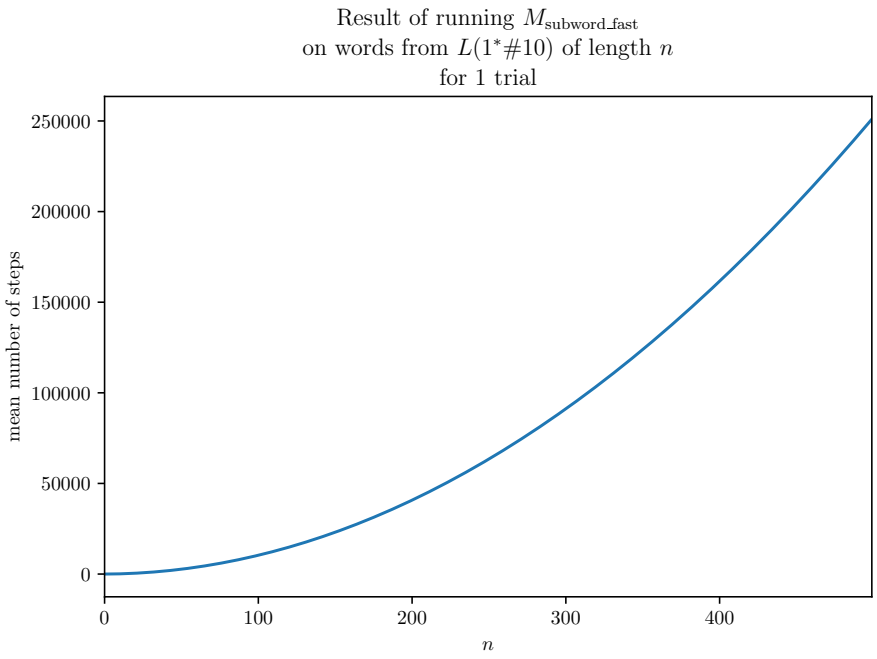


Other types of input words were also tested; please refer to the plots in section 8.4, as well as the CSV data in `data/subword-*.csv`.

3.4.4 An improvement: $M_{\text{subword\_fast}}$

Upon looking at the findings, another Turing machine, $M_{\text{subword\_fast}}$ was created that would be more optimal. Instead of only verifying letters in the left-to-right direction, the machine would also verify letters in the opposite direction. This machine has about twice as many states. It about halves the number of transitions required, but does not change the complexity class (still $\mathcal{O}(n^2)$).

The plot below shows this machine's plot.



Similarly, the other plots are available in section 8.5, and the CSV data is in `data/subword_fast-*.csv`.

4 The nondeterministic Turing machine N_{repeat}

The nondeterministic Turing machine N_{repeat} should recognize the language $\{ww \mid w \in \{0,1\}^*\}$.

4.1 Description

High-level description Read the first letter. Then move right along the tape and 'split' the machine into two machines at each cell that contains the same value as the initial cell. Each machine will then compare letters by going back and forth and marking off the letters. One of these machines will accept once all letters are compared and we reach the end of tape. Namely the machine that chooses the middle of the tape as the 'pivot'.

5 Testing

5.1 `stacscheck` tests

The project passes all `stacscheck` tests, as can be seen in the screenshot below which depicts the final few lines of the output.

```
* 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
126 out of 126 tests passed
```

5.2 Custom automated tests

To automate the testing of the Turing machines that were not covered by `stacscheck`, a custom testing tool was developed. This tests the $M_{\text{binaryunary}}$, M_{subword}, and $M_{\text{subword\_fast}}$ machines. The tool allows two types of tests: tests that verify that the machine correctly accepts and rejects, and tests that verify that the tape output is correct.

To execute these tests, run `make tests` from the `src/` folder. The screenshot below shows that the Turing machines passed all tests.

```
python3 -m tests.run_tests
Running make...
Running binaryunary tests on ./binaryunary.tm...
  PASS tests/binaryunary/accept/long1.tape
  PASS tests/binaryunary/accept/long2.tape
  PASS tests/binaryunary/accept/long3.tape
  PASS tests/binaryunary/accept/medium1.tape
  PASS tests/binaryunary/accept/one1.tape
  PASS tests/binaryunary/accept/one2.tape
  PASS tests/binaryunary/accept/one3.tape
  PASS tests/binaryunary/accept/simple1.tape
  PASS tests/binaryunary/accept/simple2.tape
  PASS tests/binaryunary/accept/simple3.tape
  PASS tests/binaryunary/accept/simple4.tape
  PASS tests/binaryunary/accept/zero1.tape
  PASS tests/binaryunary/accept/zero2.tape
  PASS tests/binaryunary/accept/zero3.tape
  PASS tests/binaryunary/reject/empty.tape
Running subword tests on ./subword_fast.tm...
  PASS tests/subword/accept/empty_subword1.tape
  PASS tests/subword/accept/empty_subword2.tape
  PASS tests/subword/accept/empty_subword3.tape
  PASS tests/subword/accept/empty_subword4.tape
  PASS tests/subword/accept/empty_subword5.tape
  PASS tests/subword/accept/empty_subword6.tape
  PASS tests/subword/accept/hash.tape
  PASS tests/subword/accept/long1.tape
  PASS tests/subword/accept/long2.tape
  PASS tests/subword/accept/long3.tape
  PASS tests/subword/accept/medium1.tape
  PASS tests/subword/accept/medium2.tape
  PASS tests/subword/accept/medium3.tape
  PASS tests/subword/accept/medium4.tape
  PASS tests/subword/accept/ones.tape
  PASS tests/subword/accept/same1.tape
  PASS tests/subword/accept/same2.tape
  PASS tests/subword/accept/same3.tape
  PASS tests/subword/accept/short1.tape
  PASS tests/subword/accept/short2.tape
  PASS tests/subword/accept/short3.tape
  PASS tests/subword/accept/zeros.tape
  PASS tests/subword/reject/empty.tape
  PASS tests/subword/reject/hashes1.tape
  PASS tests/subword/reject/hashes2.tape
  PASS tests/subword/reject/hashes3.tape
  PASS tests/subword/reject/hashes4.tape
  PASS tests/subword/reject/not_a_subword1.tape
  PASS tests/subword/reject/not_a_subword2.tape
  PASS tests/subword/reject/not_a_subword3.tape
  PASS tests/subword/reject/not_a_subword4.tape
Running subword tests on ./subword.tm...
  PASS tests/subword/accept/empty_subword1.tape
  PASS tests/subword/accept/empty_subword2.tape
  PASS tests/subword/accept/empty_subword3.tape
  PASS tests/subword/accept/empty_subword4.tape
  PASS tests/subword/accept/empty_subword5.tape
  PASS tests/subword/accept/empty_subword6.tape
  PASS tests/subword/accept/hash.tape
  PASS tests/subword/accept/long1.tape
  PASS tests/subword/accept/long2.tape
  PASS tests/subword/accept/long3.tape
  PASS tests/subword/accept/medium1.tape
  PASS tests/subword/accept/medium2.tape
  PASS tests/subword/accept/medium3.tape
  PASS tests/subword/accept/medium4.tape
  PASS tests/subword/accept/ones.tape
  PASS tests/subword/accept/same1.tape
  PASS tests/subword/accept/same2.tape
  PASS tests/subword/accept/same3.tape
  PASS tests/subword/accept/short1.tape
  PASS tests/subword/accept/short2.tape
  PASS tests/subword/accept/short3.tape
  PASS tests/subword/accept/zeros.tape
  PASS tests/subword/reject/empty.tape
  PASS tests/subword/reject/hashes1.tape
  PASS tests/subword/reject/hashes2.tape
  PASS tests/subword/reject/hashes3.tape
  PASS tests/subword/reject/hashes4.tape
  PASS tests/subword/reject/not_a_subword1.tape
  PASS tests/subword/reject/not_a_subword2.tape
  PASS tests/subword/reject/not_a_subword3.tape
  PASS tests/subword/reject/not_a_subword4.tape

Summary:
  77 out of 77 tests passed
```

6 Reproducibility

All CSV data and plots referenced in this report can be reproduced independently to verify the validity of the findings. The following commands are available for that (executed from the `src/` directory:

- `make data` will run the trials on all Turing machines and (over)write the data in the `data/` directory).
- `make plots` will generate plots for all CSV files located in the `data/` directory and place them in the same directory in the PNG format (note that this requires `matplotlib` and `latex`). This will also regenerate the `text/appendices.tex` file.
- `make report` will regenerate the entire report and output the PDF file to `report.pdf`. This also requires `latex` to be installed (specifically the `pdflatex` utility).

7 Conclusion

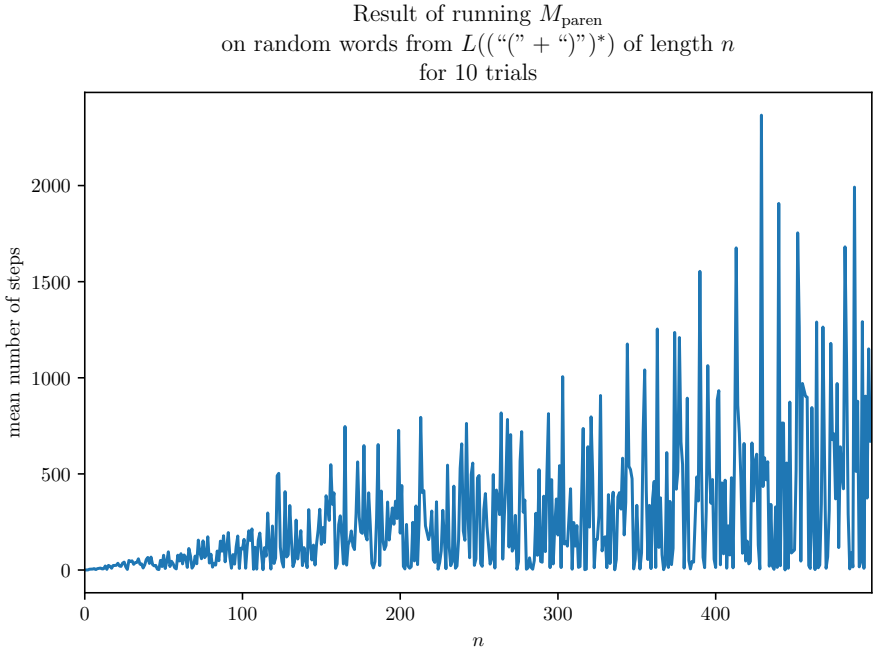
In the course of this practical, I learnt a lot about the internals of Python and the inefficiencies associated with such dynamically typed languages. I also experienced different classes of complexity, especially the binary to unary converter in $\mathcal{O}(4^n)$ that realistically could only run for input of size $n < 15$

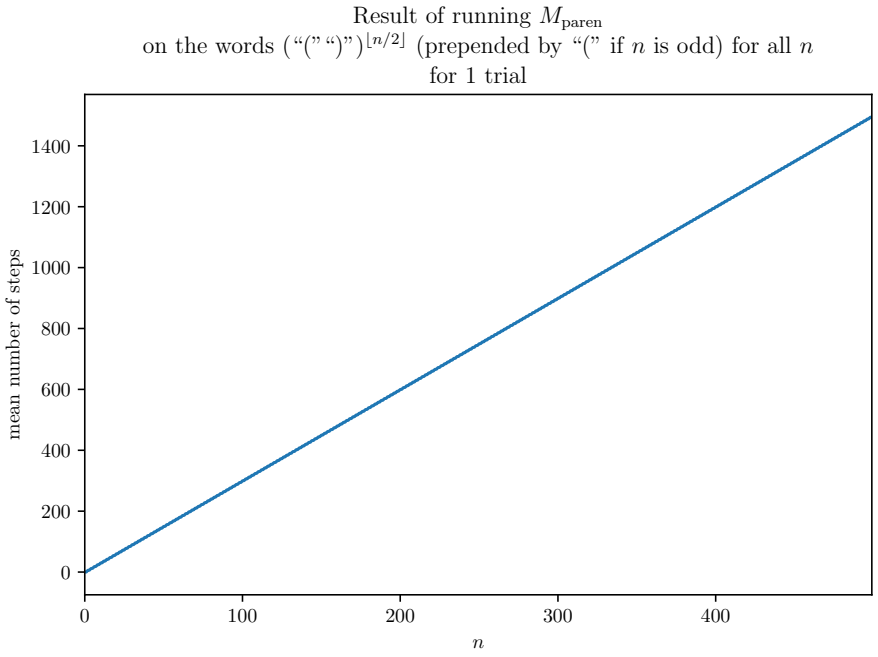
I also gained a better appreciation for the operations and methods in Turing-complete programming languages that we take for 'granted'. Their implementation on a bit-by-bit level can be quite tedious.

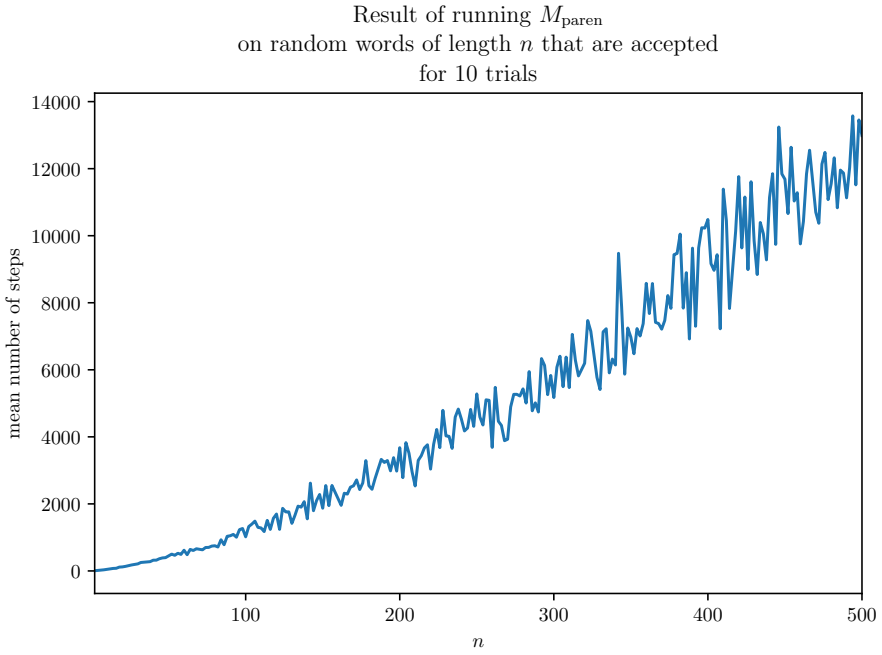
Some interesting areas for further inquiry could be to find further optimisations for the Turing machines, like $M_{\text{subword\_fast}}$ is for M_{subword}. Especially the $M_{\text{binary\_unary}}$ machine contains some redundancies. For this machine, it would be interesting to prove whether or not it is possible to construct a Turing machine for that problem that is of lower complexity than $\mathcal{O}(4^n)$ (or even $\mathcal{O}(2^n)$).

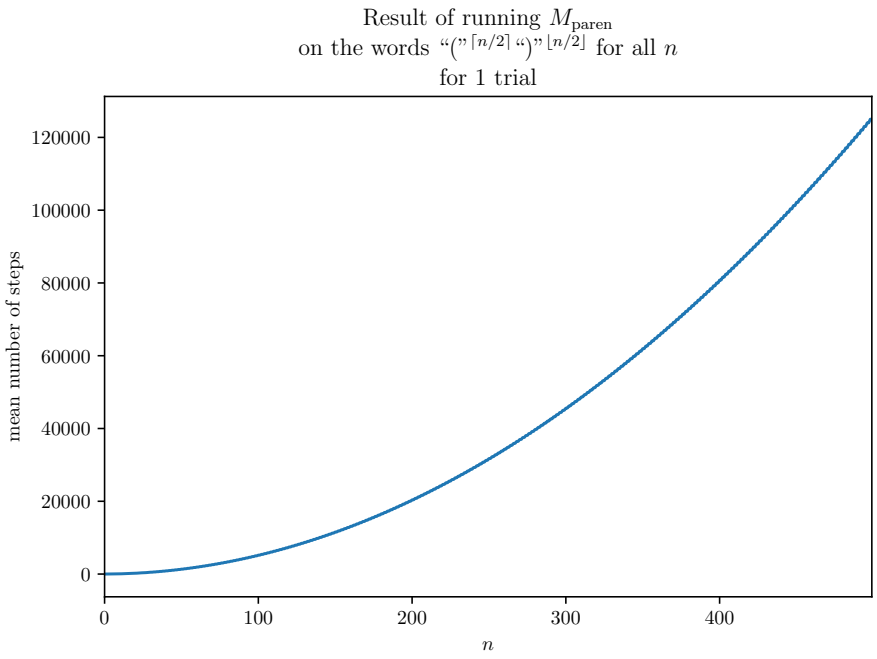
8 Appendices

8.1 Plots for the `paren` machine



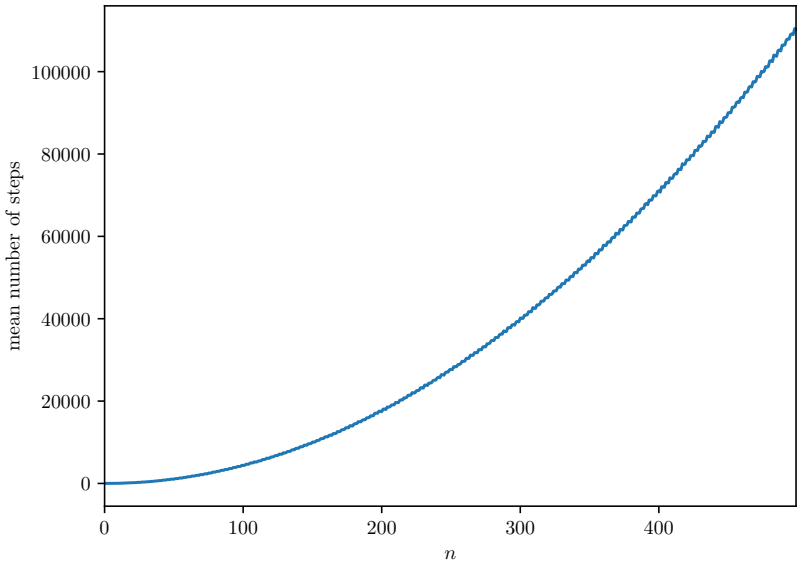

Result of running M_{paren}
on the words ("(" ")")$^{\lfloor n/2 \rfloor}$ (prepended by "(" if n is odd) for all n
for 1 trial
mean number of steps
0
200
400
600
800
1000
1200
1400
0
100
200
300
400
n


Result of running M_{paren}
on random words of length n that are accepted
for 10 trials
mean number of steps
0
2000
4000
6000
8000
10000
12000
14000
100
200
300
400
500
n

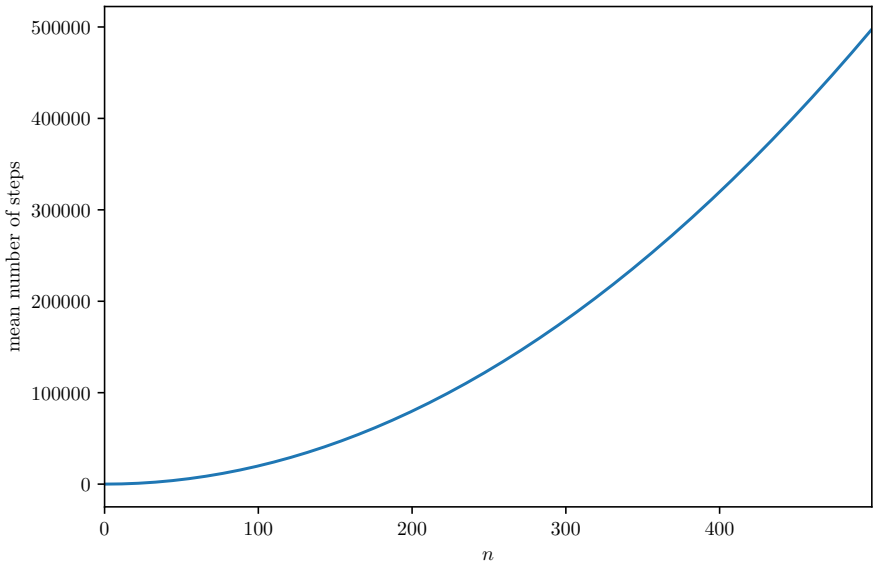

Result of running M_{paren}
on the words "("$^{\lceil n/2 \rceil}$")"$^{\lfloor n/2 \rfloor}$ for all n
for 1 trial
mean number of steps
120000
100000
80000
60000
40000
20000
0
0
100
200
300
400
n

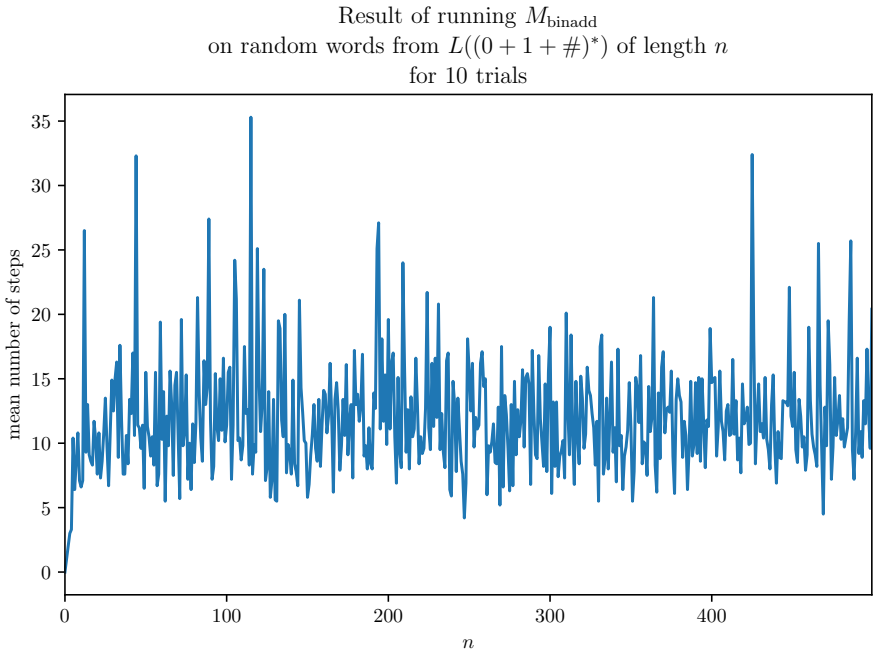
8.2 Plots for the `binadd` machine

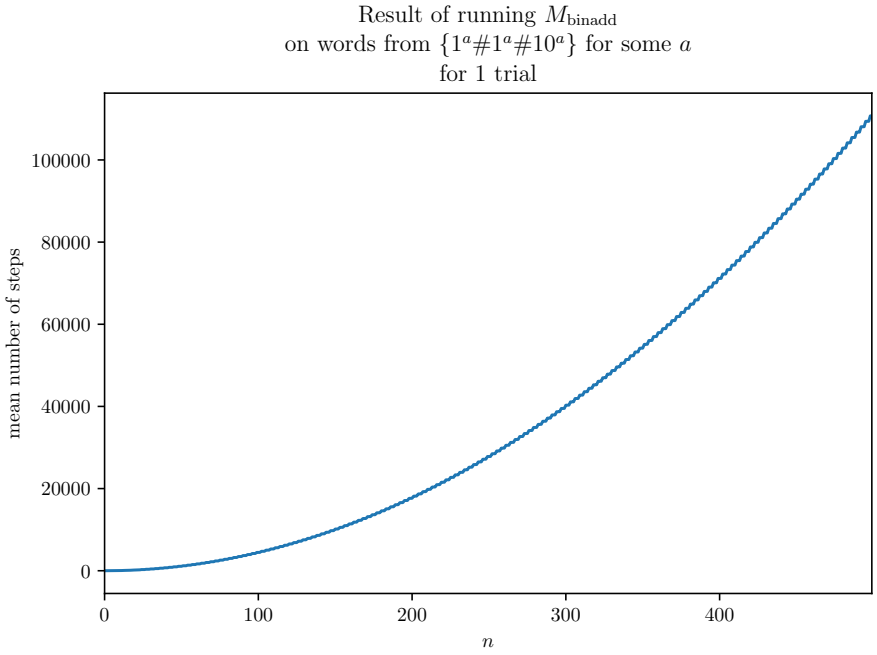
Result of running M_{binadd}
on random words from $L(\{0+1\}^*\#\{0+1\}^*\#\{0+1\}^*)$ of length n that are accepted
for 10 trials

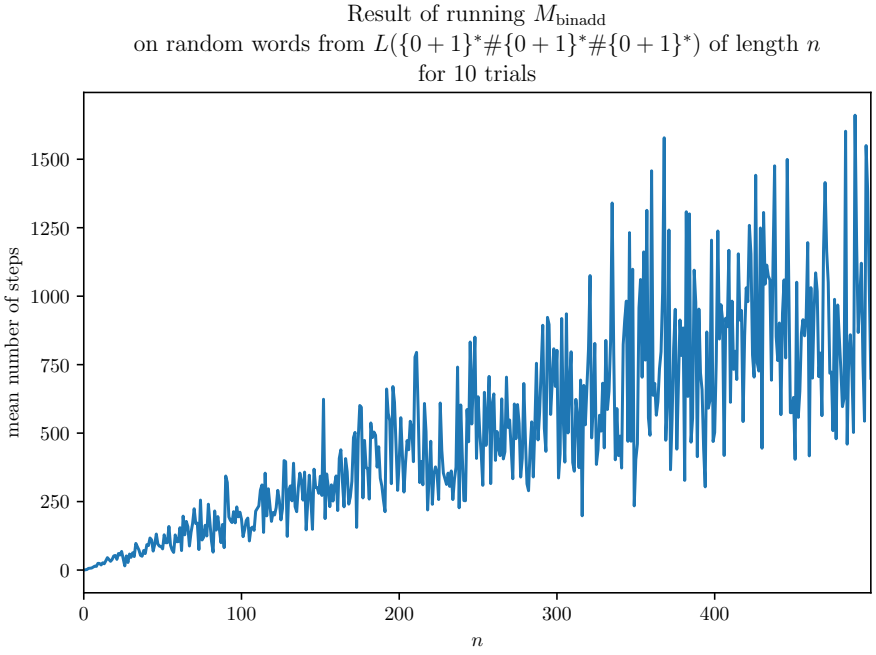


Result of running M_{binadd}
on words from $L(\#0^*\#)$ of length n
for 1 trial

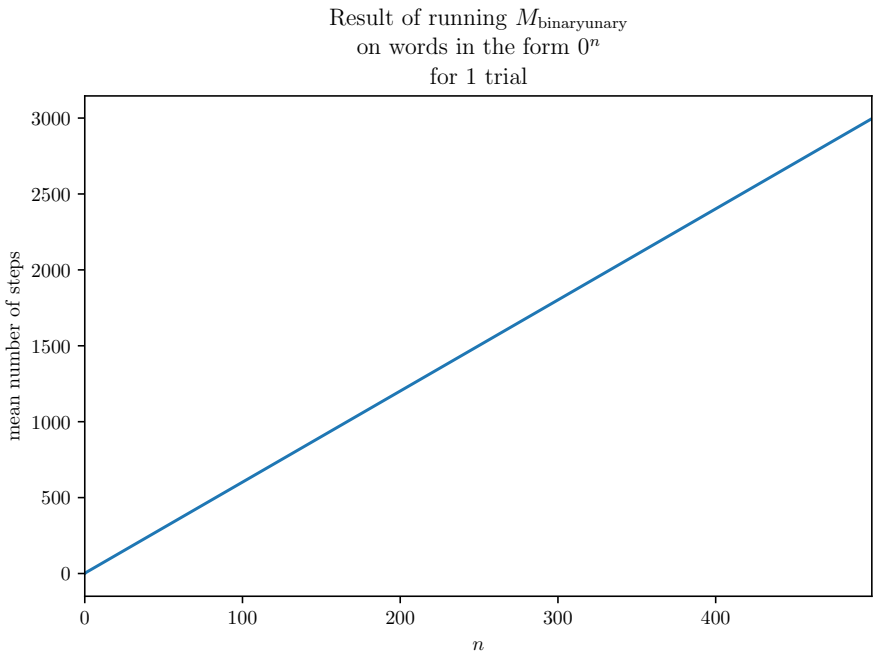


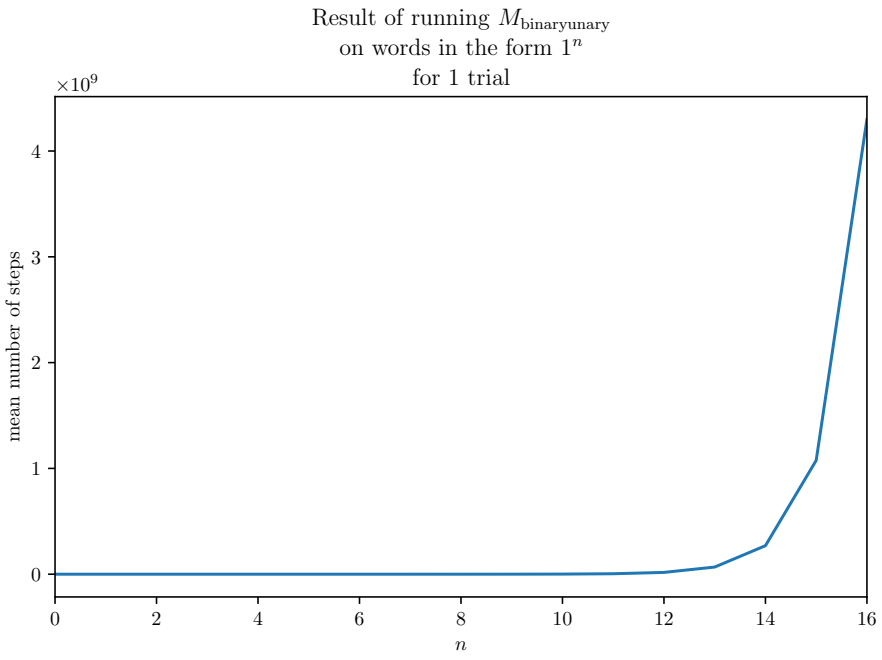

Result of running M_{binadd}
on random words from $L((0+1+\#)^*)$ of length n
for 10 trials
mean number of steps
0
5
10
15
20
25
30
35
0
100
200
300
400
n

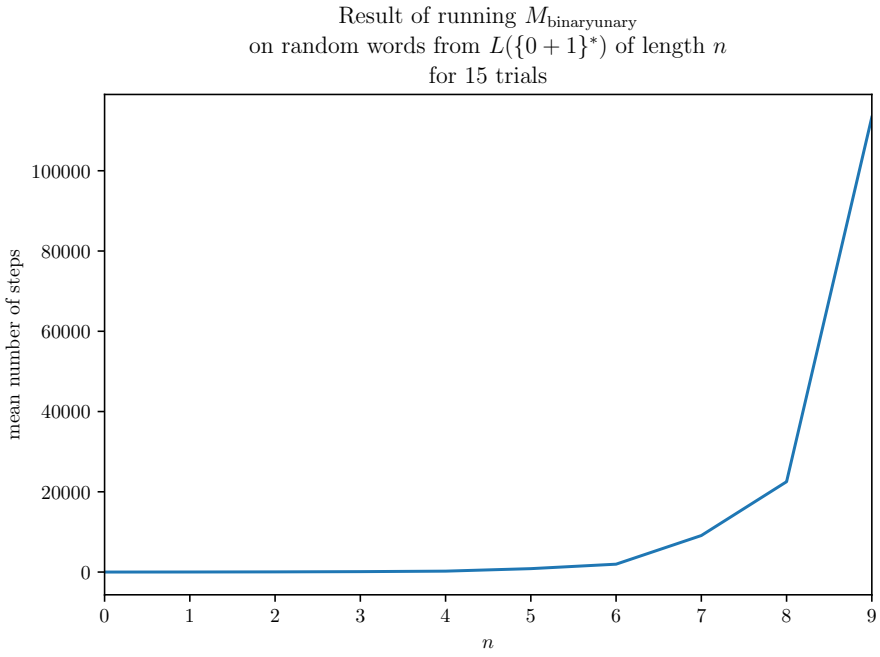

Result of running M_{binadd}
on words from $\{1^a\#1^a\#10^a\}$ for some a
for 1 trial
mean number of steps
0
20000
40000
60000
80000
100000
0
100
200
300
400
n


Result of running M_{binadd}
on random words from $L(\{0+1\}^*\#\{0+1\}^*\#\{0+1\}^*)$ of length n
for 10 trials
mean number of steps
0
250
500
750
1000
1250
1500
0
100
200
300
400
n

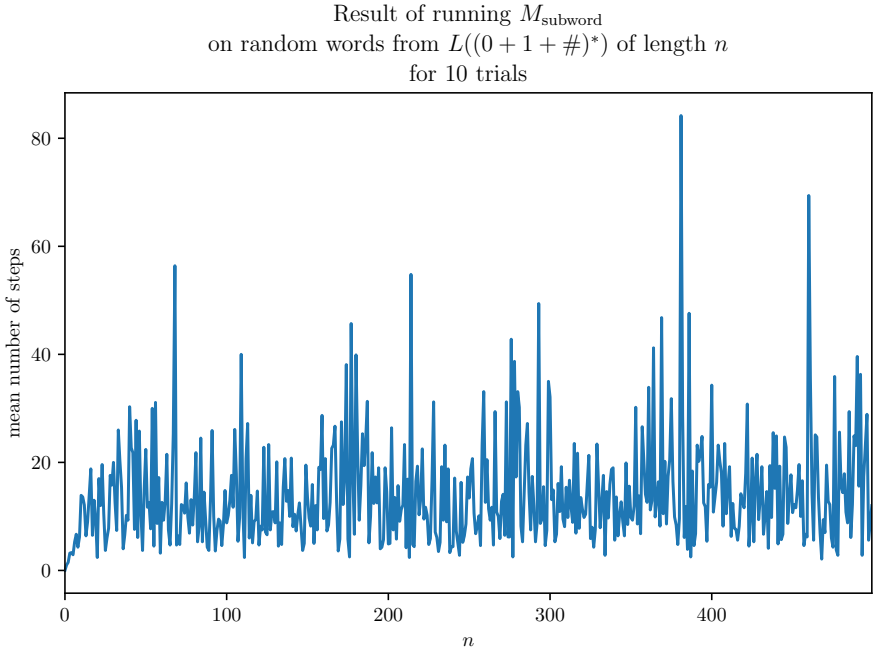
8.3 Plots for the `binaryunary` machine

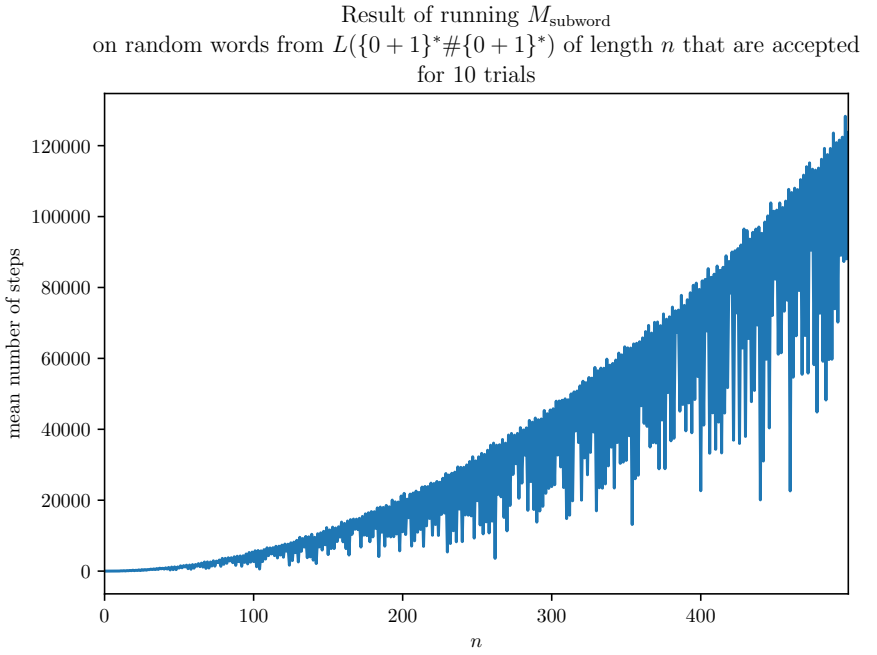


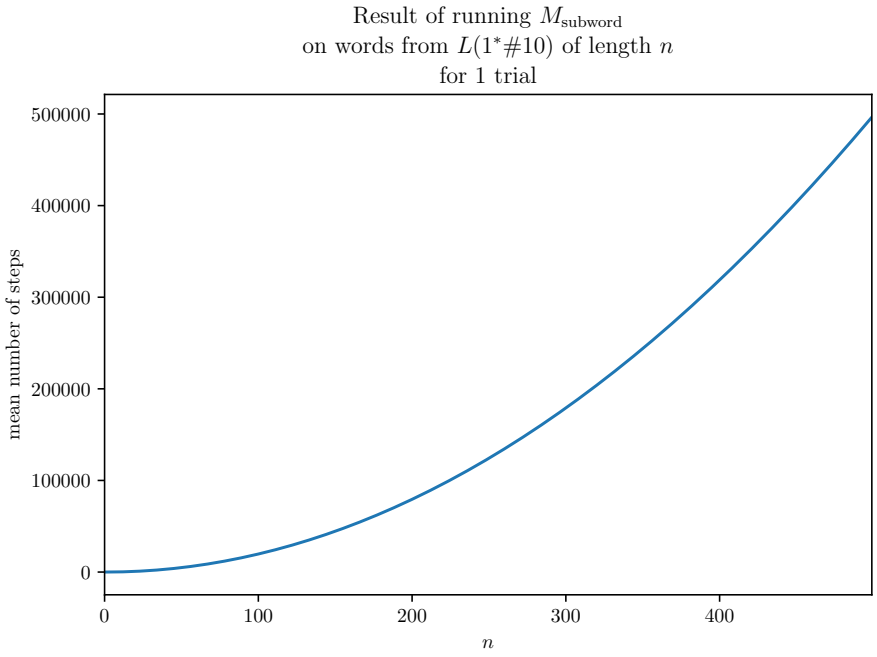


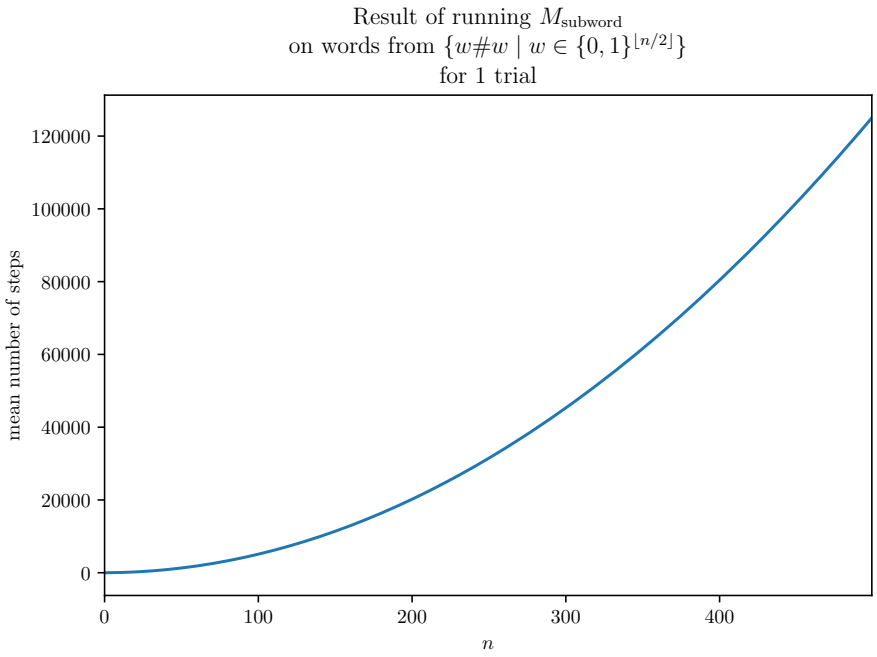

Result of running $M_{\text{binaryunary}}$
on random words from $L(\{0+1\}^*)$ of length n
for 15 trials
mean number of steps
100000
80000
60000
40000
20000
0
0
1
2
3
4
5
6
7
8
9
n

8.4 Plots for the `subword` machine

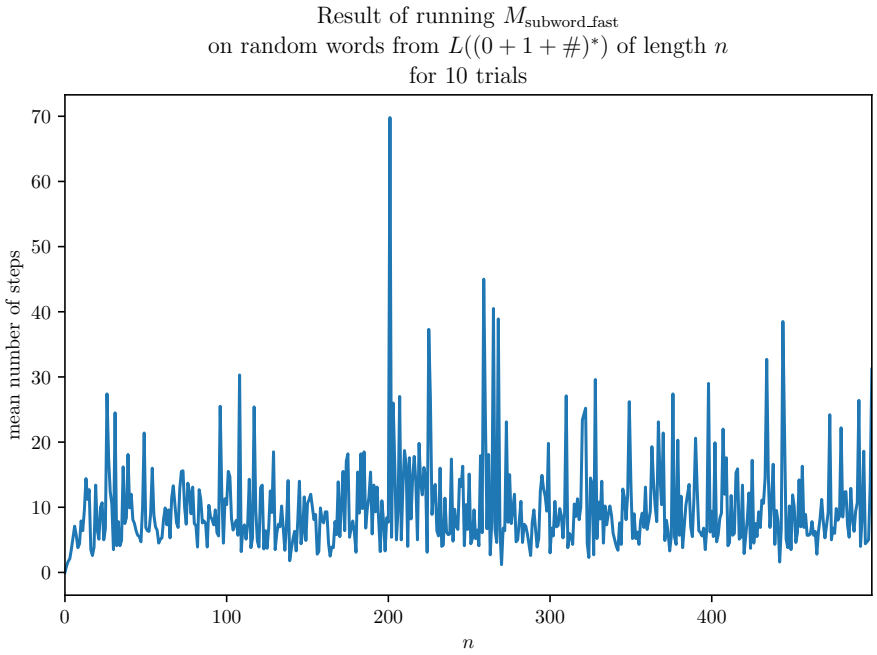


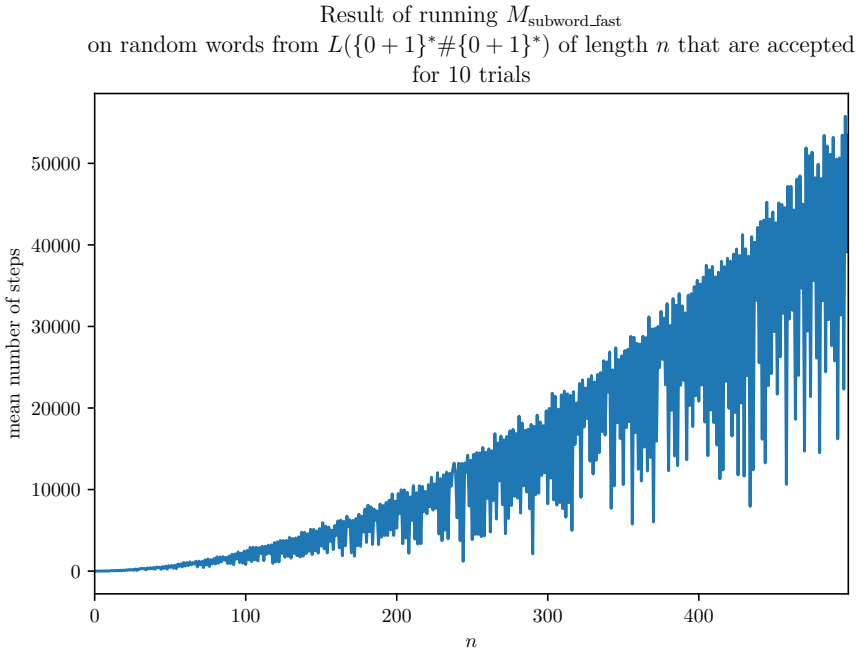


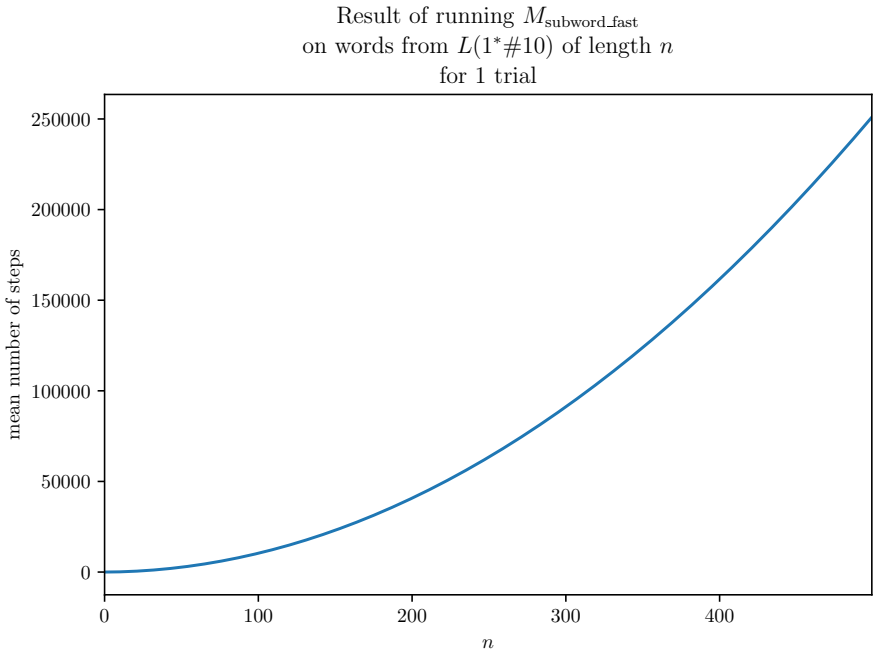

Result of running M_{subword}
on words from $L(1^*\#10)$ of length n
for 1 trial
mean number of steps
500000
400000
300000
200000
100000
0
0
100
200
300
400
n

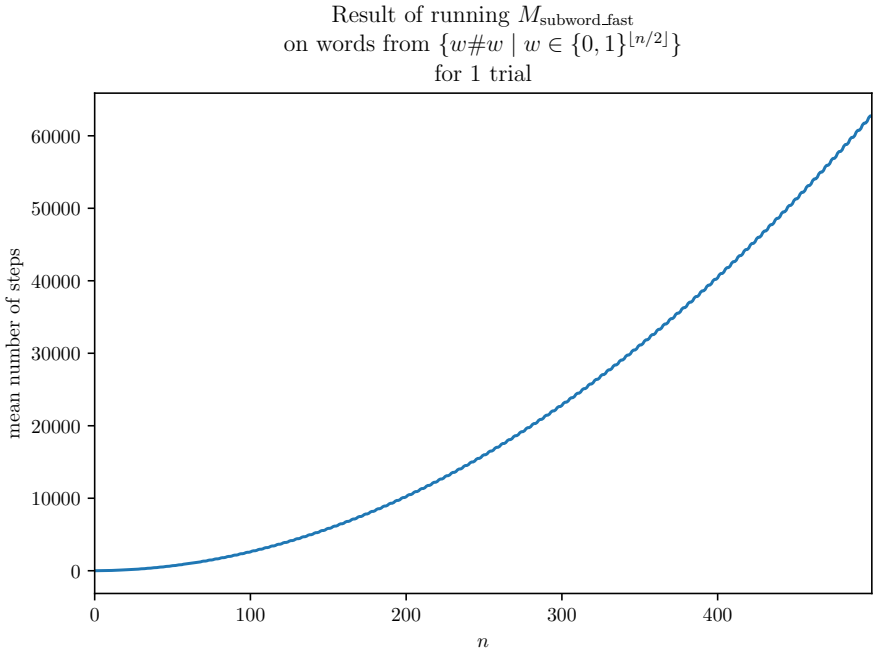

Result of running M_{subword}
on words from $\{w\#w \mid w \in \{0,1\}^{\lfloor n/2 \rfloor}\}$
for 1 trial
mean number of steps
120000
100000
80000
60000
40000
20000
0
0
100
200
300
400
n

8.5 Plots for the `subword_fast` machine






Result of running $M_{\text{subword\_fast}}$
on words from $L(1^*\#10)$ of length n
for 1 trial
mean number of steps
250000
200000
150000
100000
50000
0
0
100
200
300
400
n


Result of running $M_{\text{subword\_fast}}$
on words from $\{w\#w \mid w \in \{0,1\}^{\lfloor n/2 \rfloor}\}$
for 1 trial
mean number of steps
60000
50000
40000
30000
20000
10000
0
0
100
200
300
400
n