# INF2080
# Oblig 2

**Deadline:** March 24, 2017

## Hand-in and deadline

Hand in a *.tar.gz archive containing the *.txt files encoding your Turing machines. The TM encoding files must be named as specified in the problems. Hand in your delivery using Devilry. Deadline is **March 24, at 23:59**.

NOTE: UiO's websites do not support upper case letters in file names. Therefore, the files you download may need to be renamed (universaltm.java to UniversalTM.java and m2.txt to M2.txt).

## UniversalTM

In this exercise we will implement a Turing Machine simulator. You can write your own simulator, or modify the one we provide as needed. If you use our code, please tell us[1] about any bugs you find.

You will not be required to hand in any code. Instead, we are interested in your encoding of a selection of Turing Machines. The following example is an encoding of the TM $M_2$ (also found in M2.txt) from Sipser's Example 3.7 and Figure 3.8 (pp. 171 and 172).

```
5
0 0 1 _ R
1 _ -2 _ R
1 x 1 x R
1 0 2 x R
2 _ 4 _ L
2 x 2 x R
2 0 3 0 R
3 x 3 x R
```

[1] group teachers or Daniel: danielup@ifi.uio.no

```
3 0 2 x R
4 _ 1 _ R
4 x 4 x L
4 0 4 0 L
```

The first line contains a single number $n$, which is the number of states (except accept and reject states) in $M_2$. The states are numbered form 0 to $n-1$. The accept state is -2 and the reject state is -1.

The remaining lines encode the transition function $\delta$, with each line representing a tuple in $\delta$. By default, $\delta(q, a) = \langle -1, \_, L \rangle$ for every state $q$ and symbol $a$. That is, go to reject state, write blank, and move left. The line

```
q a p b D
```

replaces the default value of $\delta(q, a)$ with $\langle p, b, D \rangle$. We use underscore (_) to represent blank, since space is used to separate values in the encoding file.

Avoid non-ASCII symbols to avoid file encoding dependency. The safest is to use only numbers (0–9), non-accented English letters (a–z and A–Z), and the underscore (_).

The Turing Machines you make must be single tape, deterministic TMs, that work on a tape that is infinite to the right (and not to the left), that is, they cannot read to the left of start.

# Problem 1

Encode the TM $M_1$ (Example 3.9 and Figure 3.10 in Sipser), and test your encoding with several input strings. Save your encoding in a file named `M1.txt`.

$$L(M_1) = \{w\#w \mid w \in \{0,1\}^*\}.$$

# Problem 2

Encode the TM $M_{3.8a}$ described in problem 3.8a in Sipser, and test your encoding with several input strings. Save your encoding in a file named `M3-8a.txt`.

$$L(M_{3.8a}) = \{w \mid w \text{ contains an equal number of 0s and 1s}\}.$$

# Problem 3

For this problem you can choose either part a or b, or attempt both. We recommend you try part b first, and fall back on part a if you find b too difficult.

## Problem 3a

Encode the TM $M_{3.8b}$ described in problem 3.8b in Sipser, and test your encoding with several input strings. Save your encoding in a file named `M3-8b.txt`.

$$L(M_{3.8b}) = \{w \mid w \text{ contains twice as many 0s as 1s}\}.$$

## Problem 3b

Encode the TM $M$ (Example 3.23 in Sipser), and test your encoding with several input strings. Save your encoding in a file named `M.txt`.

$$L(M) = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}.$$

We encode a graph $G$ as follows:

- Begin string with `#`.
- Encode every vertex as `v<binary vertex number>` and separate vertices with `#`. A vertex can be marked by, for example, capitalizing `v`, or replacing the preceding `#`.
- Encode every edge as `e<bvn>:<bvn>`, separate with `#`.

The graph in Sipser, figure 3.24 has the following encoding:

```
#v1#v10#v11#v100#e1:10#e1:11#e1:100#e10:11#
```

Since the graph is undirected, the order within the edges are irrelevant.

Hint: When checking if the edge `e100:101` is in fact an edge from `v100`, we replace the `#` preceding `v100` by `$` to indicate that this is the active state, and do the same to the active edge. We then compare the first state number of the active edge to the state number of the active state, by comparing bit by bit. We replace every 1 with x and every 0 with y so that we know how far we have gotten, and so that we can restore the numbers afterwards.