INF 4130 Exercise set 1

Problem 1 (Coding of instances)

As described in the lecture, we model real word problems mathematically to study or solve them. When we want to write down, or code, a problem instance so that it can be given as input to an algorithm (computer program, Turing machine), or presented to another person, we need to decide on a suitable format. We describe the problem instance as a string.

a) Numbers can be written in a variety of alphabets. Compare the approximate (big-O) lengths of natural number codes in decimal, binary and unary alphabet.

(Important notions: Coding, string lengths, exponential vs. linear difference.)

Answer 1

Example, 14:

14_U = 11111111111111 (Unary) $\quad |14_U| = 14$
14_B = 1110 (Binary) $\quad |14_B| = \lfloor \log_2 14 \rfloor + 1 = 4$
14_D = 14 (Decimal) $\quad |14_D| = \lfloor \log_{10} 14 \rfloor + 1 = 2$
14_H = E (Hexadecimal) $\quad |14_H| = \lfloor \log_{16} 14 \rfloor + 1 = 1$

For a number N, let $|N_D| = n$. That is, n is the **length** of the decimal coding of the number, the number of decimal digits. What is then the length of N_B, N_H and N_U?

The largest number with n decimal digits is 10^n-1, which is $O(10^n)$. Therefore

$|N_B| \leq \log_2 10^n = n \log_2 10 = O(n)$
$|N_H| \leq \log_{16} 10^n = n \log_{16} 10 = O(n)$
$|N_U| \leq 10^n = O(10^n)$

It does not matter if we encode numbers in any radix greater than 1. The length will be roughly the same. Unary encoding, on the other hand, is not a sensible encoding because we get an exponential increase in length. The Unary encoding is exponentially longer than the Decimal encoding. (Or vice versa: The length of the Decimal encoding of number N is the logarithm of N, while the length of the Unary encoding is N.)

Problem 2 (HAMILTONICITY)

Recall that a formal language is the set of YES-instances for the corresponding (decision) problem. We can write, or code, all instances of a problem with a suitable format that we choose.

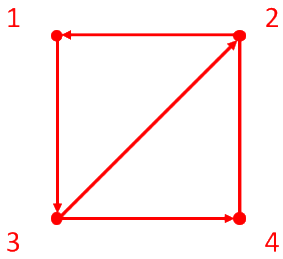
a) Show how HAMILTONICITY can be represented as a formal language over the alphabet {0, 1}. Discuss the relationship between the length of the representation and the number of nodes and edges in the input graph.

(Important notions: strings as a formalization of problem instances, formal languages as a formalization of problems. Relationship between the code lengths and the number of elements in the input.)

Answer 2

Let G = (V,E) be a directed graph. V is the the set of vertices, and E is the set of edges. The graph is Hamiltonian if it has a Hamiltonian cycle i.e. a simple cycle that includes all the vertices of G. To show that HAMILTONICITY can be represented as a formal language over the alphabet {0, 1}, we must show that every graph can be encoded as a string in the alphabet Σ = {0, 1}. Then $L_{HAMILTONICITY}$ = {x : The graph represented by x has a Hamiltonian cycle}

Take the following graph as an example (the graph is Hamiltonian, but that is not important):



How can we represent that graph?

Idea 1

The simplest way to represent a graph is to list its nodes and edges. The above graph would then be written like:

$$<1, 2, 3, 4>,<(1, 2),(2, 3),(3, 4),(4, 1),(1, 3)>$$

We don't have to represent the parenthesizes directly, but we have to separate the list of nodes from the list of edges, say by a ';'. If we use a binary representation of numbers, then we need four symbols: '0','1', ',' and ';'. These can be represented by 00, 01, 10 and 11 respectively using our two symbol alphabet. If there are n edges, every number takes at most $2 \log_2 n$ symbols to represent. There are at most n^2 edges. The length of the representation will then be at most:

$$2n(\log_2 n + 1) + 2n^2(\log_2 n + 1) = O(n^2 \log n) = O(n^3) = O(p(n))$$

The important point is that the length of the representation is polynomial in n.

(Question: Could we have used unary representation here and still had a polynomial length?)

Idea 2

A graph can be represented as a 2-dimensional matrix. The graph in the example above will then look like:

	1	2	3	4
1	F	F	F	T
2	T	F	F	F
3	T	T	F	F
4	F	F	T	F

This is easily represented as a string in 0, 1 by putting one row after the other and using one bit to represent a boolean value. The length of the representation will be

$$n^2 = O(n^2) = O(p(n)).$$

Problem 3 (Turing machines)

We use the Turing machine as our definition of an algorithm. It is a very simple model, but the simplicity makes it easy to define the notion of running time / time complexity.

a) Construct a Turing machine which recognizes the language over the alphabet {0, 1} which consists of the strings of one or more 0's only. What is the time complexity of your Turing machine?

b) Show how the above construction can be modified into a machine which recognizes the language L = {1k0k | k = 0, 1, 2, ...} (words in L consist of k zeros followed by k ones). How has the time complexity of the machine changed?

(Important notions: Turing machine as a formalization of "algorithm" and "solution". The language recognized by a Turing machine. The complexity of a Turing machine.)

Answer 3

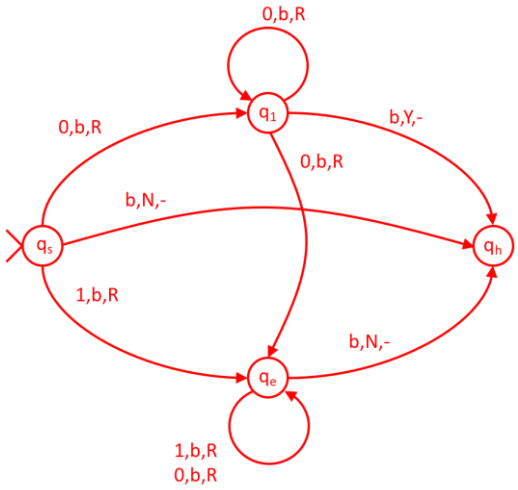
a) A TM that recognizes all strings in the alphabet Σ = {0, 1} which consists of the strings of one or more 0's only is presented in then table below. A dash ('-') means that the read/write head doesn't move. The time complexity is O(n) where n is the length of the input string.

Σ = {0, 1}, Γ = {0, 1, b, Y, N}, Q = {s, q_1, q_e, h}

state	0	1	b
0	(q_1, b, R)	(q_e, b, R)	(h, N, –)
q_1	(q_1, b, R)	(q_e, b, R)	(h, Y, –)
q_e	(q_e, b, R)	(q_e, b, R)	(h, N, –)

L = {x | x ∈ 0+}

The Turing machine can also be drawn like this (the triples indicate the symbol read, the symbol written and the movement of the read/write head):



b) Let Γ = {0, 1, $, #, b}. We can now mark out the first "1" with a $, then the first "0" with a #. We then go back and forth, marking the same number of 1's and 0's. The TM in figure 4.3 does this. The time complexity function is $O(n^2)$.

Σ = {0, 1}, Γ = {0, 1, $, #, b}, Q = {s, q_1, q_2, q_3, h}

state	0	1	$	#	b
s	(q_1, $, R)	(h, N, –)	–	(q_3, #, R)	(h, Y, –)
q_1	(q_1, 0, R)	(q_2, #, L)	–	(q_1, #, R)	(h, N, –)
q_2	(q_2, 0, L)	–	(s, $, R)	(q_2, #, L)	–
q_3	(h, N, –)	(h, N, –)	–	(q_3, #, R)	(h, Y, –)

$L = \{x | x \in 0^k1^k, k = 0,1,2,3,...\}$

Problem 4 (Universal Turing Machines) [Difficult]

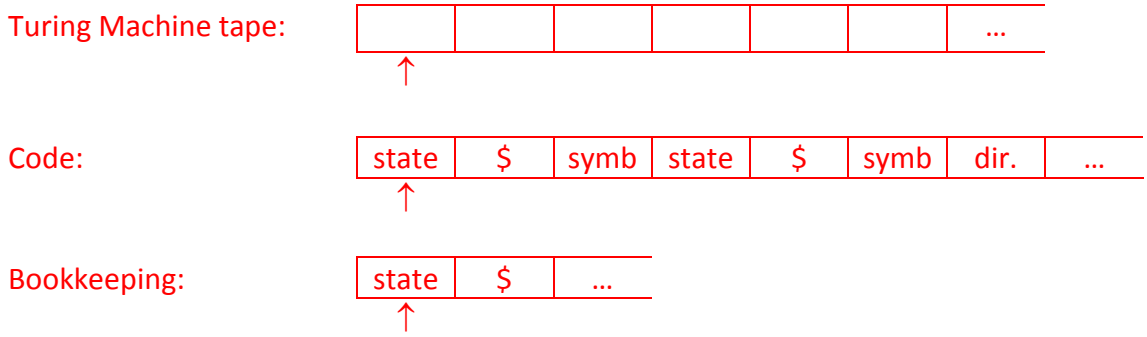
There is an obvious difference between a Turing machine and an ordinary computer: A standard Turing machine executes a single, hard coded, algorithm, while an ordinary computer takes in both an algorithm (program) and input data and runs the program on the input data.

a) Show that there is a Turing machine called Universal Turing machine (UTM) which takes a TM code M and a string I as input and then does the same as the machine M would when started on input I.

(Important notion: UTM)

Answer 4

We use three tapes for the Universal Turing Machine (UTM), one for the input and output, another one for storing the input machine code M, and the third one for book keeping, i.e. for keeping track of the state of the simulated machine:



This is OK because a machine with three tapes can be simulated by amachine that has only one tape. To encode the input TM, we need a way to represent the transition function. One simple way is to represent it as a list of 5-tuples on the form (state,read symbol,new-state, print-symbol,direction). We must also have special encodings of the special states (s and h). The UTM starts with the encoding of s on tape 3 and the input on tape 1.

Each computational step consists of:

1. Search on tape 2 for the tuple containing the state of tape 3 and the input
2. symbol of tape 1 as the two first elements,
3. copying the new state to tape 3.
4. copying the print symbol to tape 1 and
5. moving the head of tape 1 according to the encoded direction of tape 2.

The UTM halts when the new state is one of the halt states.