CS Games 2015 — Université de Sherbrooke

Theoretical Computer Science Competition

There are many facets to *Theoretical Computer Science*. As a discipline it uses advanced techniques from mathematics and logic with the aim of establishing theoretical foundations for studying rigorously all aspects related to computing, in particular algorithms and programming languages. It provides concepts and theories to capture the essence, in algorithmic and descriptive terms, of any system from specification to provably-correct implementation. More generally, it is concerned with formal models, methods and all techniques of description and analysis that are needed to investigate fundamental issues about information and information processing. The questions of the theoretical computer science competition have been formulated in this perspective.

Instructions:

- You only need paper and pencils.
- Answers too long can be written on separate sheets.
- Questions are not in any particular order (except the last question that is related to question 4).
- Answers should be clear (understandable by the examiner), precise (without errors), complete (all problem solving steps are present) and concise (solving method is as short as possible).
- You cannot ask questions during this competition.
- You have 3 hours. Good luck!

Name (1): _____

Name (2): _____

Team: _____

Question	Score	Points
1		10
2		10
3		10
4		10
5		10
6		10
7		10
8		10
9		10
Total		90

1. (10 points) THEORY IS WHEN YOU KNOW EVERYTHING BUT NOTHING WORKS. PRACTICE IS EVERYTHING WORKS BUT NO ONE KNOWS WHY. What is the mathematical theory behind each of the following programming language?

Lisp (or your favorite functional programming language) _____

Occam (or your favorite concurrent programming language) _____

Prolog (or your favorite logic programming language) _____

SQL (a query programming language) _____

Yacc (or your favorite compiler-compiler "programming language") _____

2. (10 points) COMBINATORIAL MATHEMATICS AS AN ESSENTIAL COMPONENT IN THE ANALYSIS OF ALGORITHMS. A Prüfer string is a string of length $n-2$ over an alphabet with n symbols. An interesting result about Prüfer strings is that they provide a bijection between the set of labeled trees with vertices $\{1, 2, \ldots, n\}$ and the set of strings of length $n-2$ over $\{1, 2, \ldots, n\}$.

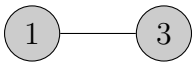
   The following algorithm constructs a labeled tree with n vertices from a Prüfer string $c = c_1 c_2 \cdots c_{n-2}$, where $1 \leq c_i \leq n$. In this algorithm, d_i is initialized with the degree of each node in the tree that the string c represents (steps 2–7).

   `Input` a Prüfer string $c = c_1 c_2 \cdots c_{n-2}$
   `Output` a tree (V, E) with $V = \{1, 2, \ldots, n\}$

0. `let` $E = \emptyset$
1. `let` $c_{n-1} = n$
2. `for` $i = 1$ `to` n `do`
3. `let` $d_i = 1$
4. `end for`
5. `for` $i = 1$ `to` $n - 2$ `do`
6. `let` $d_{c_i} = d_{c_i} + 1$
7. `end for`
8. `for` $i = 1$ `to` $n - 1$ `do`
9. `let` $j = \min(k \mid d_k = 1)$
10. `let` $E = E \cup \{(j, c_i)\}$
11. `let` $d_j = 0$
12. `let` $d_{c_i} = d_{c_i} - 1$
13. `end for`

   The following example shows how the Prüfer decoding algorithm is applied to the string $c = [1, 2, 6, 5, 1, 8]$. The array d is initialized to $[3, 2, 1, 1, 2, 2, 1, 2]$. Note that for a given string c, the degree of a node i (the number of edges incident to the node i) is one more than the number of times i appears in c.

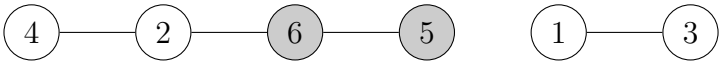
$j = \min(3, 4, 7) = 3$ and $c_1 = 1$, thus add $(3, 1)$.



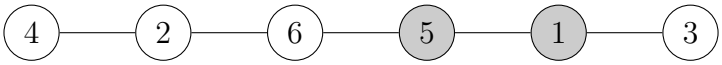
$j = \min(4, 7) = 4$ and $c_2 = 2$, thus add $(4, 2)$.

$j = \min(2, 7) = 2$ and $c_3 = 6$, thus add $(2, 6)$.

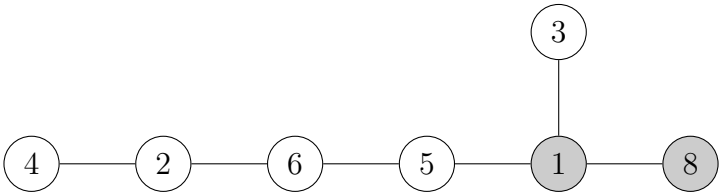
$j = \min(6, 7) = 6$ and $c_4 = 5$, thus add $(6, 5)$.



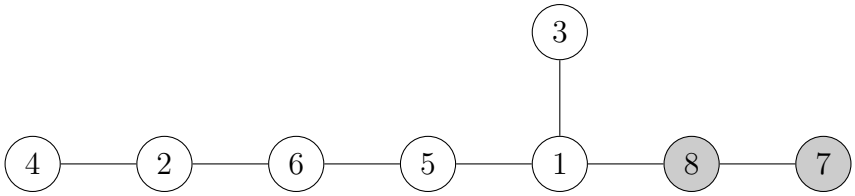
$j = \min(5, 7) = 5$ and $c_5 = 1$, thus add $(5, 1)$.



$j = \min(1, 7) = 1$ and $c_6 = 8$, thus add $(1, 8)$.



$j = \min(7) = 7$ and $c_7 = 8$, thus add $(7, 8)$.



(5 points) Provide a computational complexity analysis of the previous decoding algorithm and express your final results in the usual asymptotic notations Ω and O (you can annotate the program if necessary).

(2 points) Which complexity class the problem of decoding a Prüfer string belongs to?

Prüfer strings are attractive because they can be easily manipulated in a genetic algorithm that processes trees. For instance, there are many applications where it is necessary to find an optimal spanning tree with specific structural properties (e.g., an optimal tree in which the degree of each node in less than a given integer $k > 0$). The underlying problem in these applications is the generation of the Prüfer strings.

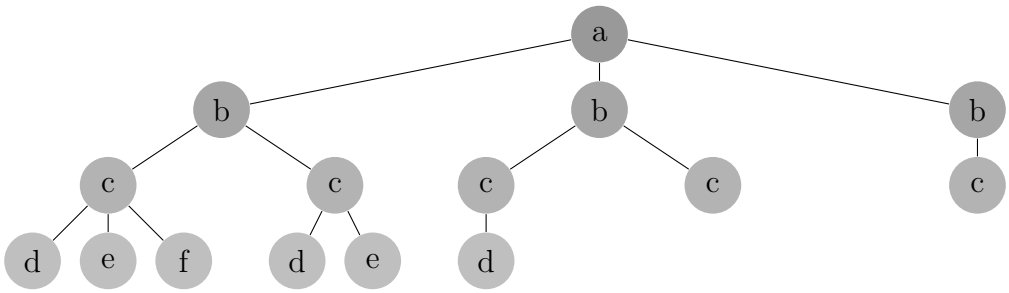
(3 points) Suppose that an application requires to generate all Prüfer strings and, for each string, find the corresponding tree in order to determine if it is the potential optimal one w.r.t. to some criteria. Such an algorithm is a *brute-force* algorithm. Which complexity class the problem solved by this algorithm belongs to?

3. (10 points) HA, THE FASCINATING QUANTUM COMPUTER POSSIBILITIES! Everybody, even your grandmother, have heard that quantum computers will be able to solve complex problems that are beyond the power of even tomorrow's most powerful conventional supercomputers. In particular, your grandmother wants your opinion, as computer scientist or software engineer, about that. Fortunately, your grandmother is bright. She wants a persuasive answer with respect to the notable question "P = NP or P $\neq$ NP ?". In other words, could quantum computers solve *hard* problems as easily as *tractable* problems? Transcribe the answer that you would give to your grandmother onto the space provided below.

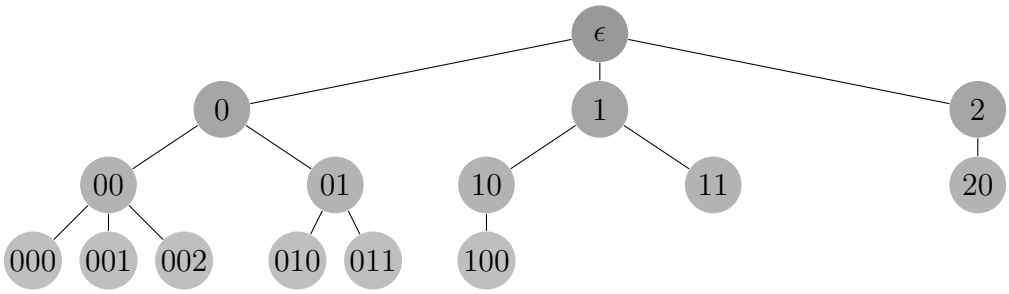
4. (10 points) FORMAL DEFINITIONS AS THE STARTING POINT OF A THEORY—THE THEORY OF TREE AUTOMATA! Given an alphabet Σ, a Σ-valued tree t is specified by its set of nodes (the "domain" $\mathrm{dom}(t)$) and a valuation of the nodes in the alphabet Σ. The domain $\mathrm{dom}(t)$ is included in the set $\{0, 1, \ldots, k-1\}^*$. Formally, a *k-ary Σ-valued tree* is a map $t : \mathrm{dom}(t) \to \Sigma$, where $\mathrm{dom}(t)$ is a nonempty set, closed under prefixes, which satisfies

$$wj \in \mathrm{dom}(t),\ i < j \Rightarrow wi \in \mathrm{dom}(t).$$

As an example, the following tree is a ternary $\{a, b, c, d, e, f\}$-valued tree. In particular, $\mathrm{dom}(t) \subseteq \{0, 1, 2\}^*$ (recall: ϵ denotes the empty word).



t



$\mathrm{dom}(t)$

($2\frac{1}{2}$ points) For any finite k-ary Σ-valued tree t, does the domain $\mathrm{dom}(t)$ (the set of nodes) define a regular language over $\{0, 1, \ldots, k-1\}$? Justify your answer.

($2\frac{1}{2}$ points) The *outer frontier* of a k-ary Σ-valued tree t contains the elements $wi \notin \mathrm{dom}(t)$, where $w \in \mathrm{dom}(t)$ and $i < k$. Formally,

$$\mathrm{fr}^+(t) = \{wi \notin \mathrm{dom}(t) \mid w \in \mathrm{dom}(t) \wedge i < k\}.$$

What is the outer frontier of the tree depicted in the previous page (you can draw the outer frontier in the tree depicted in the previous page)?

($2\frac{1}{2}$ points) Provide a formal definition for the frontier of a k-ary Σ-valued tree t, denoted $\mathrm{fr}(t)$.

($2\frac{1}{2}$ points) Provide a definition for a path from the root of a k-ary Σ-valued tree t to a given node that belongs to $\mathrm{fr}(t)$.

5. (10 points) THE FAMOUS *halting problem*! The halting problem is stated as follows. Given any program, we want to decide if it ever halts. Of course, we can run the program for a while, but what if the program has not halted after a billion years. Turing proved there is no program to solve the halting problem.

   (1 point) Who is Turing?

   The main arguments of its proof are the following. Suppose by way of contradiction that such a program P exits. Then, we can modify P to produce a new program P' that does the following. Giving another program Q as input, P'

   - runs forever if Q halts given its own code as input, or
   - halts if Q runs forever given its own code as input.

   Feed P' its own code as input. By the two conditions above, P' will run forever if it halts, or halt if it runs forever. Therefore, P' — and by implication P — cannot have existed in the first place.

   In conclusion the halting problem is undecidable.

   (6 points) Now, consider an *autopoiesis* program, that is, a program capable of reproducing itself. As for the halting problem, one can ask if such a program exists. The simplest case is a program that does nothing, except self printing its own code. Do you think that such a program could be written in your favorite programming language? If yes, what it looks like? If no, what are your arguments that prove the contrary?

(3 points) What do you need to change to your previous answer if the program does something useful?

6. (10 points) The language XML as a formal language. XML is a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable$_{\text{Wikipedia}}$. Here are some rules with short explanations from the XML 1.0 specification: rules [16] and [17] that define *processing instructions* and rules [18] to [21] that define *CDATA sections*.

```
Processing Instructions
[16]  PI       ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
[17]  PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
```

> PIs are not part of the document's character data, but MUST be passed through to the application. The PI begins with a target (PITarget) used to identify the application to which the instruction is directed. The target names "XML", "xml", and so on are reserved for standardization in this or future versions of this specification. The XML Notation mechanism may be used for formal declaration of PI targets. Parameter entity references MUST NOT be recognized within processing instructions.

```
CDATA Sections
[18]  CDSect  ::= CDStart CData CDEnd
[19]  CDStart ::= '<![CDATA['
[20]  CData   ::= (Char* - (Char* ']]>' Char*))
[21]  CDEnd   ::= ']]>'
```

> Within a CDATA section, only the CDEnd string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form; they need not (and cannot) be escaped using "<" and "&". CDATA sections cannot nest.

(2 points) Rules [16] to [21] are production rules of a context-free grammar written in an extended notation (strings of characters between single quotes are terminal symbols). The goal of this question is to define *processing instructions* **or** *CDATA sections* with the aid of a regular expression, since the operators concatenation, Kleene closure ($*$), disjunction ($|$), zero or one ($?$) and difference ($-$) are also used to write regular expressions, except one! Which one?

(5 or 8 points) Give a regular expression for *processing instructions*, that is, the language defined by rule [16] (5 points) or, if you want to accumulate more points, give a regular expression for *CDATA sections*, that is, the language defined by rules [18] to [21]. To simplify the problem, ignore `PITarget`, that is, only consider the following rule:

```
Processing Instructions
[16']  PI ::= '<?' (S (Char* - (Char* '?>' Char*)))? '?>'
```

Assume that `Char` represents any character acceptable by XML and `S` represents a space.

7. (10 points) STACKS, STACKS, STACKS, ... ! A *pushdown automaton* is a sextuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q is a finite set of states, Σ a finite set called the input alphabet, Γ a finite set called the output alphabet, $q_0 \in Q$ the initial state, $F \subseteq Q$ a set of final states, and δ a transition function from $Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\})$ to subsets of $Q \times (\Gamma \cup \{\epsilon\})$. There are several acceptance conditions defined on such automata.

   (2 points) Consider the language $L = \{a^i b^j c^i d^j \mid i, j \geq 0\}$. Show by using the pumping lemma that the language L is not a regular language.

(2 points) By using intuitive arguments explain why this language is not a context-free language.

(2 points) What modifications should be done to *pushdown automaton* so that the language L will be accepted by an instance of your new kind of machine?

(2 points) Give the complete formal definition of the instance of the machine that accepts L, that is, give the set of states, the alphabets, the initial state, the final states (with the acceptance conditions), the transition function, etc.

(2 points) Is your machine has the computational power of a Turing machine?

8. (10 points) No, a proof is a proof. What kind of a proof? It's a proof. A proof is a proof, and when you have a good proof, it's because it's proven!$_{\text{ex-prime minister Jean Chrétien}}$ Zoe has been murdered, and Ada, Boris, and Charlie are suspects. Ada says she did not do it. She says that Boris was the victim's friend but that Charlie hated the victim. Boris says he was out of the town the day of the murder, and besides he didn't even know the girl. Charlie says he is innocent and he saw Ada and Boris with the victim just before the murder. Assuming that everyone—except possibly for the murderer—is telling the truth, use the resolution to solve the crime.

   The idea of resolution is simple. If we know that P is true or Q is true and we also know that P is false or R is true, then it must be the case that Q is true or R is true. Formally, given a clause (a cause is a set of literals representing their disjunction, where a literal is an atomic sentence or the negation of an atomic sentence) containing a literal ϕ and another clause containing the literal $\neg\phi$, we can infer the clause consisting of all literals of both clauses without the complementary pair.

$$\frac{\begin{array}{ll}\Phi & \text{with } \phi \in \Phi \\ \Psi & \text{with } \neg\phi \in \Psi\end{array}}{(\Phi - \{\phi\}) \cup (\Psi - \{\neg\phi\})}$$

   As an example, consider the following deduction. The first premise asserts that either P or Q is true. The second premise states that either P is false or R is true. From these premises, we can infer by resolution that Q is true or R is true.

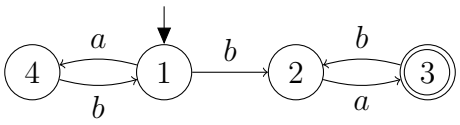
1.	$\{P, Q\}$	a premise
2.	$\{\neg P, R\}$	a premise
3.	$\{Q, R\}$	from 1,2

9. (10 points) INFINITE WORDS, INFINITE TREES! Regular expressions and context-free grammars define languages of finite words. The latter can be accepted by finite automata and pushdown automata on the same alphabet, respectively. This means that for a given regular expression, there is a finite automaton that accepts the language defined by the regular expression (the converse is also true). Similarly, for a given context-free grammar, there is a pushdown automaton that accepts the language generated by the context-free grammar (the converse is also true).

   For a given alphabet Σ, the set of finite words is denoted Σ^*. This question concerns automata that accept a subset of the set of *infinite* words denoted Σ^ω. Don't be afraid about that, in particular the ω (omega) symbol. For example the word $ababaaa\ldots = ababa^\omega$ is the infinite word that starts with $abab$ followed by an infinite sequence of a. You can imagine the difference with the set of finite words $ababa^*$.

   (2 points) Describe in your native language the word $(ab)^*(ba)^\omega$.

   An ω-automaton is an automaton that recognizes infinite words. More specifically, an ω-automaton is a quintuple $(Q, \Sigma, \delta, q_0, Acc)$, where Q is a finite set of states, Σ a finite set called the alphabet, $q_0 \in Q$ the initial state, Acc the acceptance component, and $\delta : Q \times \Sigma \to 2^Q$ the transition function. There are several acceptance components. The simplest one is called the *Büchi acceptance*: $Acc = F \subseteq Q$ and a word $\alpha \in \Sigma^\omega$ is accepted by such an automaton if and only if at least one of the states of F has been visited *infinitely often* during the recognition of α. The following automaton accepts the words $(ab)^*(ba)^\omega$ ($q_0 = 1$ and $F = \{3\}$).



(2 points) Construct an ω-automaton with *Büchi acceptance* that recognizes the language

$$L := \{\alpha \in \{a, b\}^\omega \mid \alpha \text{ ends with } a^\omega \text{ or ends with } (ab)^\omega\}.$$

By analogy to the translation of regular expressions into finite state automata (finite word automata), logic formulas can be translated into ω-automata (infinite word automata). These formulas must be, however, written in a *linear temporal logic*, which is useful to model the dynamics of systems (or "software programs") that never terminate.

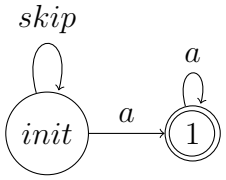
This logic is built up from atomic propositions $p_1, p_2, \ldots, p_n$ by means of Boolean connectives, (and, or, negation), the unary temporal operators X ("next"), F ("eventually"), G ("always"), and the binary operator U ("until"), which is not considered hereafter.

A propositional temporal logic φ is interpreted in ω-sequences $\alpha \in (\{0, 1\}^n)^\omega$. In a word of length n formed of 0 and 1, 0 in position i means that p_i does not hold and 1 in position i means that p_i holds. Furthermore, in an infinite sequence formed of words

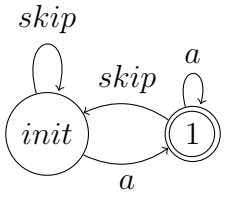
of length n, $\alpha[i]$ refers to the ith word and $\alpha[i, \omega]$ refers to the *omega*-sequence from the ith word. The satisfaction relation $\alpha \models \varphi$ is defined inductively by the following clauses (we only provide a subset of clauses):

$\alpha \models p_i$	iff $\alpha[0]$ has a 1 in its ith component;
$\alpha \models \varphi_1 \wedge \varphi_2$	iff $\alpha \models \varphi_1$ and $\alpha \models \varphi_2$;
$\alpha \models \mathrm{X}\varphi$	iff $\alpha[1, \omega] \models \varphi$ (φ holds next time);
$\alpha \models \mathrm{F}\varphi$	iff $\exists i \geq 0$ such that $\alpha[i, \omega] \models \varphi$ (φ eventually holds);
$\alpha \models \mathrm{G}\varphi$	iff $\forall i \geq 0$, $\alpha[i, \omega] \models \varphi$ (φ always holds).

There is an extraordinary tool on the Web that translates any linear temporal logic formula into an ω-automaton. It is called LTL2BA. For example, the formula "F (G a)", which is a canonical formula for a *persistence* property, is translated into the following automaton:



The formula "G (F a)", which is a canonical formula for a *response* property, is translated into the following automaton:



(2 points) Draw the ω-automaton that corresponds to the following formula:

$$(\mathrm{X}\ a) \wedge (\mathrm{X}\ \mathrm{X}\ b) \wedge (\mathrm{X}\ \mathrm{X}\ \mathrm{X}\ \mathrm{G}\ c)$$

(2 points) A tree automaton is an automaton that recognizes sets of infinite trees. In any infinite k-ary Σ-valued tree, every node has k direct successors and branches never terminate. What is then $\mathrm{dom}(t)$ (see question #4)?

(2 points) What is a path in an infinite k-ary Σ-valued tree?

(End of questionary)