CMSC485 (Section 2, Spring 2003) *Theory of Computation* Sample Problems

This document contains sample problems for this semester. The problems are supposed to be 'practical', at least to some extent. You may find some contexts where your action might be different depending on whether or not you know the answer (and why). If you are not interested in any of these problems or know how to solve these problems, there is no point for you to take this course. On the other hand, if you find some of these problems interesting and/or relevant, this course will introduce you to the Theory of Computation so that you will be able to answer not only these questions but also related ones, which might pop up in the future. The sample problems are collected from various sources. Some are adapted or re-created for this course.

1. Infinite Loop Detection

Infinite loop is one of the most common bugs in your (or anyone else') program. It would be enormously helpful if someone writes a program to detect infinite loops in a given program (as a text file). Would it be possible to write such a program?

2. Mayan Script

You found an ancient Mayan script that is supposed to indicate the location of hidden treasures. The only language resources in that language you have access to are: (1) an extensive list of synonyms and (2) a collection of sentences whose meaning are already known.

For example, let's imagine that you need to translate the following sentence (obviously, not real Mayan): "koregawakarukane" with the following resources:

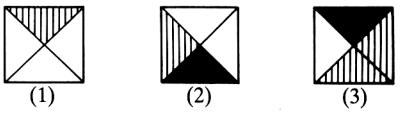
Synonym list: "korega" = "maa", "ruka" = "nnee", "neene" = "darona", etc.
Sentences whose meaning are known: "maawakanneedarona", "mosikasitarawakarukamone", etc.

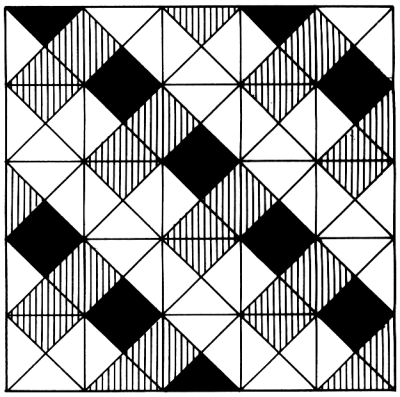
A sample session of translation by substituting synonyms would be (underlined words are replaced with *italic* words):
"korega wakarukane" → "*maa* wakaruka ne" → "maawakan*nee* ne" → "maawakanneed*arona*"

Then, we can tell that "koregawakarukane" means the same thing as "maawakanneedarona", which we can understand. If there are many sentences to analyze and the synonym list is extensive, we will naturally think of doing the task with a computer. Would it be possible to write a program to solve a problem of this kind?

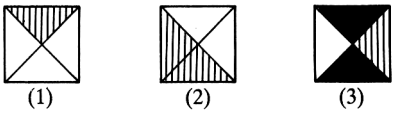
3. Floor Tiling

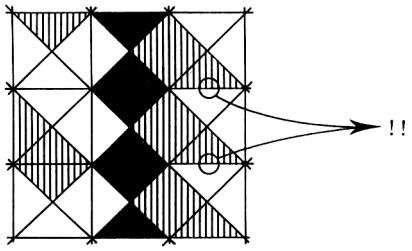
Suppose that you are searching for a house to live in and have found a one you like, except that none of the floors are covered. So, you decided to buy the house any way and finish the entire floor with a variety of tiles available at the Home Depot. You will be using *n* types of *square* tiles with distinct patterns. To demonstrate your esthetic sense, you will match the edges of the tiles as shown below (in this case, using three types of tiles). [Images from *Algorithmics* by David Harel]





Note that if you choose a wrong set of tiles, you may never be able to demonstrate your esthetic capability, as shown below.





Would it be possible to write a program to tell whether a given set of *n* types of square tiles can fill your floors with the above-mentioned condition?

4. Linear Programming

Consider the following problems:

- Find the cheapest combination of foods that will satisfy all your nutritional requirements
- Minimize the risk in your investment portfolio subject to achieving a certain return
- In order to mass produce complex electronic circuit boards, an almost uncountable number of holes need to be drilled. How do we route the drill (attached to a robot arm) to visit all these holes so it takes the shortest possible route and time?

All of these problem can be tackled by a technique called ‘linear programming’. Roughly, linear programming is an approach to set up *k* linear inequalities with *n* variables and find an assignment for the variables that satisfies all the inequalities.

As many problems are rather complex, researchers have been solving them mainly through case-by-case approaches. Would there be a general way to solve all of these problems?

5. Ambiguity Detection

A simple rewriting system can be used to characterize a certain language (set of strings). For example, consider the following set of rewrite rules (can be used to represent programming language syntax):

$A \rightarrow x\ A\ y$ (Rule 1)
$A \rightarrow x\ y$ (Rule 2)
$A \rightarrow x\ x\ y\ y$ (Rule 3)

Starting from the symbol *A*, we can generate all sorts of strings of the form *x...xy...y* where the number of *x*'s and that of *y*'s are the same. For example, applying Rule 1 twice and Rule 2 twice would result in an output '*xxxyyy*' as shown below:

$A \rightarrow x\ A\ y \rightarrow x\ x\ A\ y\ y \rightarrow x\ x\ x\ y\ y\ y$

One tricky point about this type of formulation is that the system can be 'ambiguous'. That is, there may be more than one way of generating the same result. For example, the above system can generate '*xxxyyy*' also by applying Rule 1 once and then Rule 3 once. Naturally, ambiguous systems could cause a lot of trouble for precise representation of a language and also for processing the language. Would there be a general way to detect whether this type of system is ambiguous?

6. Program Verification

One of the software industry's biggest concern is how to check whether their products are really correct with respect to the specification that the developer and the user both agreed. According to a variety of sources, inability to do so will cost us an incredible amount of money in the future (well, this must be already happening). As you know, *generality* is a prime concern in Computer Science. So, why don't we write a single program that could verify whether the program correctly solves a problem with respect to a given specification? But would it be possible?

7. Arithmetic System

One of the great achievements in logic is that it can formalize a wide variety of systems. For example, the mathematical system of arithmetic can be formalized in first-order logic in a consistent (roughly, meaningful) way. Its basic components include formulas such as "for any *x*, $x + 0 = x$" and "for any *x*, $x \times 0 = 0$". In this system, we can write all sorts of formulas that are representative of arithmetic. One property that a good logic system must have is that all 'true' formulas can be proven as a series simple steps. But here is a big question. Would it be really possible to prove all true formulas in the arithmetic system?

8. Hilbert's Tenth Problem

A great mathematician Hilbert once asked: whether or not a given polynomial equation with integer coefficients has a solution in integers. Is this in general solvable?

9. Program Elegance

Many programmers may be trying to write 'elegant' programs (only if they have time, which is of course never available). But could we ever tell whether a particular program is elegant?

10. Functional Programming Languages

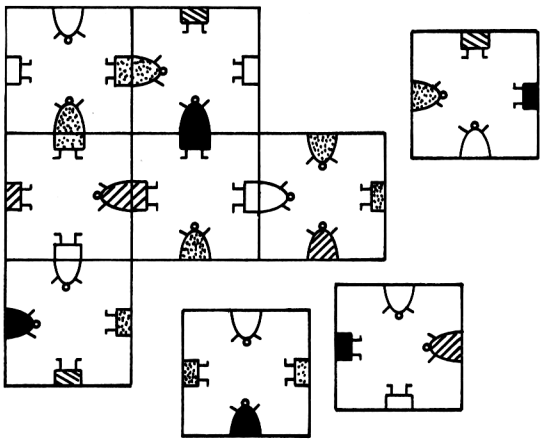
LISP/Scheme, ML, and Haskell are known as 'functional' programming languages. They differ from more popularized 'procedural' programming languages such as Basic and C. We can often observe that programs in a functional language look rather different from those in procedural languages, even for doing the same task. For example, you might never see a loop, only recursions. However, both of these language groups seem to be able to do pretty much the same thing. Could we justify that they can actually solve exactly the same class of computational problems?

11. Cryptography

Public key cryptography employs two types of keys, public and private. The basic idea behind the use of two keys is that (A) the public key can easily be obtained from the private key and (B) the private key can***not*** easily be obtained from the public key. For example, RSA cryptosystem satisfies these conditions. However, cryptology is a never ending battle. We will need to have many other cryptosystems for coming years (in fact, also right now). Then, how can we draw the line between 'easy' and 'not-easy' computation in a systematic manner?

12. Monkey Puzzle

The following is a snapshot of a game to place 9 square cards so that pictures will match (note that the cards cannot be rotated). [Image from *Algorithmics* by David Harel]



In general, we can think of a puzzle in any $n \times n$ size. How fast could we solve such a puzzle?

13. Digital Circuits

To design digital logic circuits, we need to analyze the relation between inputs and the output. For example, the following formula represents a circuit with four inputs and one output, each of which could take either 0 or 1 ('+' for OR, '·' for AND, and '' for NOT).

output = $(i_1' + i_2 + i_3 + i_4) \cdot (i_1' + i_2 + i_3 + i_4') \cdot (i_1' + i_2' + i_3 + i_4')$

Given an arbitrary formula, we can surely find out whether some combination of inputs would result in output 1, by checking all the combinations (you should be able to tell how long it would take to compute all the results for the *n*-variable case). However, we would surely hope that there is a better way. How fast could we compute the output in general?

14. Professor Assignment

It is not a simple task of assigning all the CS faculty members to all the courses offered by the CS department. Well, it is actually not that complicated at TCNJ. But what about a large English department in a giant state university? Imagine that as a part of your work study, you are asked to write a program to do the scheduling for a large department. How fast in general could your program schedule all the faculty members?

15. Knapsack Problem

There will be an excursion tomorrow. You will need to pack all of your junk food packages in your knapsack. However, you realized that not all of them would fit in the knapsack. So, you want to fit as many as possible while the total cost of your junk foods is maximized (to impress your pals). How fast could you find out the solution given an arbitrary scenario?

16. Cross-Country Interviews

Suppose that you are invited for interviews (graduate schools or industry jobs) in a number of cities in the US. Unfortunately, they do not pay for your travel. So, you will need to minimize the expense. How fast in general could you find the best way to travel all the cities?

17. CPU Register Allocation

CPU is no doubt the 'brain' of a computer. We want to cram as many things as possible in CPU's. However, we also need to consider the cost of doing so. In practice, we have to settle down at some cost-performance level. The cost-performance trade off also applies to the number of registers in a CPU. In order to find the optimal number of registers for a future CPU, we want to write a program to find the minimal number of registers for a collection of popular programs. What would be the performance of such a program?

18. Map Coloring

It has been shown that with four distinct colors, we can color any map so that the neighboring countries (or whatever political boundaries) do not share the same color. With three colors, we may or may not be able to do the same thing. How fast in general could we find out whether a map can be 3-colored?

19. Time-Space Tradeoffs

If our computers had an infinite amount of memory, we could load all the programs and run them without accessing the hard disk. This would eliminate our frustration with loading time. However, this will remain as dream. Then, we need to compromise the use of space (memory, etc.) and the processing time. Can we say anything general about the time-space tradeoff?

20. *Respectively* in English

In English, you can say something like: *a*, *b*, *c*, and *d* are the lower case of *A*, *B*, *C*, and *D*, respectively. In general, there is no limit to the number of items that are connected in this manner. What would be the simplest mechanism that can process the correspondence of multiple sequences like this?

21. Programming Language Parsing #1

Modern programming languages allow an arbitrary level of nesting. You do not need to write a LISP program to be baffled by an insane number of '(' and ')' as well as all sorts of other nesting constructions. What would be the simplest representation (i.e., that can*not* represent anything more complicated) for this type of conditions? What would be the simplest mechanism that can handle the 'nesting' property of modern programming languages? In addition, are there properties of programming languages that would require more than handling nesting?

22. Programming Language Parsing #2

Suppose that we know the answer to the previous problem. In other words, we know how to characterize the set of all valid programs in terms of program structure. Then, given an arbitrary program, can we always identify whether it belongs to the set? In other words, is parsing possible? Of course all sorts of programming languages are being parsed by all sorts of compilers and interpreters. The question here is more general because we are talking about all programming languages with, e.g., the nesting property, including those we have never seen.

23. Password Screening

When you obtained an account for a host computer, you might have been asked to choose a password that satisfies the conditions: (i) at least six characters, (ii) must include at least one symbol or numeral, (iii) must include at least one upper case character, etc. It shouldn't be difficult to check all these conditions. But in order to maintain the security of the system, this kind of screening must definitely be done by a program. What would be the simplest representation (i.e., that can*not* represent anything more complicated) for this type of conditions? What would be the simplest mechanism that can handle this and similar conditions in a systematic manner?

<End>