COMPUTER SCIENCE TRIPOS, Part II (General)
DIPLOMA IN COMPUTER SCIENCE

Mathematics for Computation Theory **(KM 1999)**

Study Guide

This course contains two rather separate parts. The first six lectures (Part A) develop a rapid course in Discrete Mathematics: an introduction to the basics of sets, functions and relations, leading to partially ordered sets, well founded relations and proof by induction. The second half of the course (Part B) applies these ideas to a specific problem, that of characterising the power of finite automata in discriminating between sequences of input symbols: what emerges, the theory of regular languages, is satisfying. Part A is Pure Mathematics, while Part B is theoretical Computer Science.

No single guide can be appropriate to all Diploma students, since there's such a big variation in mathematical background. It's also the case that some students with a good degree in continuous mathematics find the going hard, while others with little formal training accept the ideas without difficulty and can do all but the hardest examples. If you're relatively new to the ideas of Part A and you don't find it easy, then there's an excellent book which may help:

Bernard Kolman and Robert C Busby

Discrete Mathematical Structures for Computer Science (Prentice-Hall, 2nd Ed., 1987).

There are lots of diagrams and plenty of exercises with each chapter. Although it covers a lot of material that you don't need (but not everything that you do!) it should be clear from the schedules which sections are relevant.

This is a course of Discrete Mathematics, the second part Applied to computing. As is the case with all mathematics courses it's vital that you tackle lots of examples. Two example sheets will be made available as the course proceeds: the first contains a number of exercises relating to Part A, in roughly the order that the lectures will follow; the second is a list of Tripos problems that relate to both parts of the course. All of these should be accessible from the stance and notation adopted in the lectures. (Please make sure that you use the current example sheets, which are improved in a number of ways.)

As in past years I shall be giving Example Classes in connection with the course: these are not intended to replace supervisions. **There will be three classes, taking place in TP4 at 2.10 pm on Thursdays November 11th, 18th and 25th**. At these classes I shall start by covering any questions arising from the lectures or from the distributed notes, then go on to give solutions to one or more examples from the sheets (chosen according to demand, or failing that on my whim). The classes are aimed at students with less rather than more background, though experts will of course be welcome. I am prepared to help anyone who has specific problems immediately after the class, but **not** to take solutions away for correction (it's up to your supervisor to mark your work).

Please let me know about any problems with the lecture material or the example sheets.

Overall course structure

There are (many) different approaches to the theory of finite automata and regular languages, and the one that I've followed lies at the mathematical end of the spectrum. The reason for this is that my brief is to teach a course of Discrete Mathematics for use in Computer Science and in addition to cover regular languages, and it seemed natural to tie the theory to the application. Previously there were 16 lectures on Discrete Mathematics (with a separate course on regular algebra), and some of the material was included for its inherent interest rather than its relevance. The position now is quite different, there's no time for any topic that is not essential to the development.

The result is that a lot of rather desirable material has been cut. Topics whose absence is damaging include *directed graphs* (with reference to binary relations on a set), *complete partial orders* and *inductive definition through finitary rules*. The final sections of Part A develop a *Principle of Structural Induction* that is sufficient for the application to *regular algebra*, but it isn't really adequate for a more general treatment of programming language syntax and semantics. The first part of the notes was rewritten for 1998 to develop *well-founded relations*, incidentally, and you should beware of obsolete versions. A side effect is that there are a few extra pages in Part A, and I'm afraid that I haven't taken the trouble to renumber the pages of Part B correspondingly (so each Part has a page 70 . .).

The notes as distributed include a course schedule and a reading list, as well as a copy of this study guide. The first half, including the material on induction, should take exactly the first 6 lectures (though the extra pages may stretch Part A into the start of lecture 7). I shall do my best to get rid of the experts at least for the first four lectures, because their presence escalates the pace, and they tend to ask posh questions. The second half of the course (on Regular Languages) will start at 12 noon on Friday November 19th, give or take a few minutes, but it's probably a good idea to turn up to review the treatment of well-founded relations and induction.

Part A

The schedule splits 6 lectures into six sections, which are typically of roughly equal length. If you've not met *sets* before then you need to read up basic Boolean algebra, and undertake some simple exercises. Not everyone likes pictures, but if you do then getting the hang of Venn diagrams early on will give you a useful tool.

The rest of Part A is really a quick dash to set up the basic properties of order relations. Order relations play a significant role in computer science; the purpose of computation is often to gain additional information about the solution to some problem that is posed as input. A typical example might be to calculate the (irrational) root of some polynomial by an iterative technique. After running for some specified time the computation will have produced only the first however-many digits in the decimal representation of the root, but the specification of the computation (the program, if you like) is not limited in precision, instead allowing the calculation of every digit in the (infinite) decimal representation of the root. The notion of *successive approximation* is captured by an order relation.

To establish the correctness of such an iterative computation requires logic to handle the fact that the expansion of the root is infinite: some notion of *limit* is needed. *Induction* offers a tool that will help in certain cases; the final section of Part A presents a style of induction that is tailored for use in Part B.

There are lots of good text books on Discrete Mathematics, and there's no reason for you to favour the ones in the reading list particularly. I'm not aware of any introductory text that presents the material on well-founded relations that makes up the final two lectures.

Part B

The aim of these lectures is to prove important results in theoretical Computer Science fairly rigorously, using the techniques introduced in Part A. The material is not always easy, but I hope that once again it's intuitive.

The problem area that we investigate is to characterise *the language accepted by a finite automaton*. A *finite automaton M* is modelled as a sequential device with finite logic (hence a finite set Q of internal states) that responds deterministically to the symbols from some finite alphabet S by a change of state; the only observation that we may make distinguishes some subset of states as *accepting* (a green light flashes, for example).

Suppose that M is started in some nominated *initial* state. How can we identify the set of strings in S (the finite sequences whose elements are input symbols) whose application symbol-by-symbol to M will cause transition into an accepting state? It turns out that this set will always be denoted by some *regular expression over S*, as introduced in Part A. Further, given any regular expression over S, we can construct a finite automaton with input alphabet S which accepts precisely the event denoted by that regular expression.

How do we prove these statements? Perhaps the latter is the easier of the two, we use structural induction in the language of regular expressions over S; in the course of the proof we show how to construct a suitable finite automaton step-by-step via the parse tree of the given regular expression.

The first part can be regarded as an instance of a standard technique in theoretical computer science. The action of the given finite automaton may be characterised by its transition matrix M, which specifies its *operational* behaviour; we want to describe the language accepted by the automaton, thus providing *denotational* semantics for the automaton. The proof once again uses induction, in this case the familiar mathematical induction over the size of the matrix M. The algebra involved is not difficult, but it depends on manipulating matrices whose entries are sets of strings; there is an unfamiliar operator * whose properties are sometimes counter-intuitive.

The notes contain examples of finite automata to which the theory may be applied, and a further simple automaton was set as a problem in Paper 2 in 1998. There is no substitute for experimenting with *regular algebra*, which is the appropriate tool for manipulating the languages accepted by finite automata.

There are plenty of books that prove Kleene's Theorem, but only the one by John Conway (now out of print, but in some libraries) handles matrices of events in the same style as the lectures. Hopcroft and Ullman's 1979 book is excellent for the final lecture.