Complexity

Part IB, II(G) & Diploma

A C Norman, Easter Term 1999

1 Introduction

This course is about the complexity of computation. This relates to the way in which computing times grow as you solve larger and larger examples of problems. Part of the course will be illustrating how complicated things get if one tries to seek the very fastest (in terms of this ultimate growth rate) way of solving even simple problems. A middle section looks hard at an area generally accepted to be the boundary between problems that it is feasible to solve and ones that are too hard, while at the end of the course I will sketch a demonstration of how a simple looking problem may have associated costs that are beyond most people's nightmares.

This course build on material from several previous earlier sets of lectures:

Early courses: Big-O notation for describing rates of growth;

Regular Expressions: Recall that these provide a neat way of writing patterns. I will quote the results about these that I need, and will not need you to remember the associated proofs!

Turing Machines: It is important for this course that you are happy about the structure of a Turing Machine and the fact that it is a general model of computation. What may amaze you is that it is a *good* model of practical computation quite often;

Complex numbers: These get mentioned at one stage;

Modular arithmetic: This is just performing ordinary integer arithmetic but only keeping the remainder when the natural result is divided by some modulus M. It certainly arose in the Part IA Discrete Maths course when RSA encryption was introduced;

General skills in algorithms: Both design and analysis skills are involved. Various of the parts of my course amount to presentations of techniques that could almost equally have gone into an Algorithms lecture course.

The presentation here is based on four books, three of which are easy to obtain and indeed that you may own already. The last book is older and may be less readily available, so I have included the important material as part of these notes.

Because these books (and in fact many others) cover this material well you have the advantage that you can look in them and find quite large numbers of example questions some of which will be less substantial than past Tripos questions (of which there are plenty, available via the computer laboratory web pages in the usual way). When I point out that a certain part of my presentation is keyed to one of the books you should automatically assume that this means I am drawing your attention to the examples in the book at the end of the relevant section and suggestion that you try some of them. The books are:

Knuth[3]: The Art of Computer Programming is a classic set of volumes. I will take just snippets from volume 2, but I continue to believe that well-rounded computer scientists browse all the rest of that and the other volumes to ensure they have an overview of what is described;

Sudkamp[4]: Languages and Machines will provide a useful refresher for those who want to brush up on their regular expressions as well as being close to the presentation I will give of some hard problems;

Cormen at al[2]: If you have not already bought your copy of this thick book now is another excuse. I will only be using the last two chapters from it here;

Aho, Hopcroft and Ullman[1]: This was an important book in its area and covers the very hard problem I will discuss at the end of the course. I suggest you consult it after you have read all of the previous three suggestions!

Because these books are so good I will not include all the fine details of the material in these notes. Many of you will find that your supervisors will understand this material quite well and that between that and the lectures you will be happy, but in terms of precise explanation of technicality it would be very hard for me to match the clarity of exposition in any of the recommended texts.

There are parts of this course that are technically delicate. Many of the terms I introduce seem to lend themselves to informal abuse or mis-application, or at least application where the precise result that they imply has not been proved. My presentation will still try to concentrate on explaining the structure of the results and on skipping detail where I feel I reasonably can. The intent is that once you have grasped the overview you will be in a position to work through everything again adding back full formality and precision of notation. Current research areas in the area of complexity rely firmly on extreme precision of reasoning and so may re-cast some of the ideas I show you here in concise, precise mathematical formulations that could initially look a bit daunting: I hope my more informal presentation will help lead some of you into the area and that you will then enjoy it.

2 Easy

I a course on data structures and algorithms the tasks looked at are generally of around the size and messiness as "sorting". In this part of the complexity course I will show that there is quite a lot to be studied in tasks that you probably previously took for granted! In particular the issue to be investigated will be implementing integer arithmetic. It seems pretty clear to me that until we know just how fast this can be done it will hardly be possible to analyse any other algorithm at all.

A complexity-theory approach twists our intuition by asking only about growth-rates of costs. This is a sensible approach in that it allows us to ignore the constant-factor scalings associated with fast or slow computers. IN the case of integer arithmetic it will be natural to measure the size of inputs as the number of bits in a binary representation of the numbers. The challenge is then to find the sharpest possible big-O bound on the costs of basic arithmetic operations. There are three oddities:

1. The insistence on looking for the best big-O growth rate as the way of assessing algorithms sometimes causes us to look at methods where the constant (hidden in the big-O notation) is rather large even though the growth-rate is slow. Such methods may only pay off on ridiculously large inputs, and so practicality is not guaranteed;

2. The best intellectual result will often be a proof that some task will have costs that grow at least as rapidly as some function, regardless of the algorithm used to solve it. Eg we have seen elsewhere that "sorting using comparisons most cost on average at least $\log n!$ whatever method is used". Such proofs do not necessarily show us how to achieve the bound: for sorting we were lucky that merge-sort (for instance) did achieve the lower bound, but for other tasks we can end up with a mathematical bound and no clue as to how to meet it;

3. For many tasks the exact model of computer permitted when considering algorithms is somewhat problematic. Big-O notation talks about ultimate limits, while real computers have limited address ranges and limited precision arithmetic for calculating addresses. If you had to imagine sorting (say) 2^{66} numbers then think about the index arithmetic needed to address them, and note that if the numbers are only 32 or 64-bit values you are bound to have lots of repetitions. We need to abstract away from these frivolities without letting anything horrid creep in. I suppose I should give an example of something that would count as "horrid" here. Suppose we extended our idea of a computer to make it roughly like all current ones, but accepted it could have arbitrarily large memory, that all memory could be accessed in unit time and that the index arithmetic needed to calculate memory addresses had constant cost. Then it might turn out that various problems could be solved unexpectedly rapidly by packing lots of 32-bit integers into a single address register and using devious address arithmetic

and huge tables to do in effect parallel processing on all the integers. I will cheat a little in the first section of this course and use as my machine model a programming language like ML or Java and I will count each basic operation as taking unit time. I will avoid thinking too hard about the problems of addressing truly huge amounts of data. Later in the course and for other problem areas I will be able to avoid this fudge.

Because I want to look at asymptotic complexity I will tend to think of arithmetic on *big* integers. For a small range of numbers your computer may do the arithmetic in unit time already, or you could pre-compute a table of all possible calculations and instead of arithmetic you just have a look-up operation to perform.

For arithmetic I will suppose that numbers are to be represented in binary. On occasions it will be useful to clump the bits together in groups of three (for octal), four (for hexadecimal) or more (eg in practical cases thirty-two to fit with real machines) but I will suppose that my computer or programming language has some fixed width arithmetic that it can perform as an elementary operation and that anything on numbers bigger than that has to be worked out using carries etc.

First observe that addition, subtraction and comparison of n-bit numbers can be performed in time $O(n)$ using obvious algorithms. Furthermore these algorithms can not possibly be completed without being prepared to access all the bits of the inputs, and there are n of those. Thus it is not possible for there to be an algorithm with time faster than $O(n)$.

Knuth discusses some details. In particular for a theoretical standpoint one might like to represent huge numbers as linked lists (last significant part first) because then the average cost of increment and decrement operations are constant (despite the potentially long chain of carries: on average the carry does not go very far) and the cost of adding two numbers of different length is determined by the length of the shorter. Here I will imagine that whenever numbers are to be combined they are both the same length, and I will think of them as stored in an array. Doing this will not have an effect on my discussion of the costs of multiplication and division, which are the next matters for concern.

Consider the problem of multiplying two integers together, and suppose the numbers are big enough that long multiplication is called for. Primary school long multiplication forms the product of two n-digit numbers in around n^2 steps. Theoretical computer scientists ask if it is possible to do better. An easy scheme (Karatsuba) reduces the cost to around $n^{1.57}$ (and I will explain where the strange exponent comes from. The best theoretical limit know so far[1] is $n \log(n) / \log \log n$ and the fastest algorithm discovered to date has cost $n \log n \log \log n$. The gap

[1]Well this depends rather critically on the model of computer you have, and if you read Knuth hard enough you will find discussion of the possibility of linear-cost multiplication.

between these formula represents a gap in our knowledge (it is not of very great practical concern, but that does not stop it being a worry to those with tidy minds).

Karatuba's algorithm takes a pair of n-bit integers and views them as two-digit values where each new digit has $n/2$ bits. I will write this as $(a, b) \times (c, d)$. The normal way to multiply a pair of two-digit numbers would form the four products ac, ad, bc and bd. Karatsuba instead forms three products: ac, bd and $(a-b)(c-d)$. Well in the last of these you could use either either have added or subtracted, and it is really an $n/2 + 1$ bit product not an $n/2$ bit one. By subtracting and then splitting $a - b$ and $c - d$ into their signs and absolute values you can get back to using $n/2$ bit arithmetic. I leave you to sort out how to reconstruct the full product from these parts: look it up in Knuth or attend lectures if you need help!

If $M(n)$ is now the total cost of multiplying two n-bit numbers then using Karatsuba we can achieve

$$M(n) = 3M(n/2) + kn$$

for some constant k that covers the additions, subtractions and overheads. It is easy to sketch enough code to see that these costs all grow linearly. The solution to this recurrence shows $M(n) = O(n^{\log_2 3})$ which explains the funny exponent mentioned earlier. Practical experience shows that this method becomes useful for multiplying numbers that need around 10 words (eg say 300 to 600 bits) to represent them. It is thus of practical relevance. It also shows that the simple $O(n^2)$ multiplication was not optimal. It also forms a warm-up to the explanation of a much more elaborate multiplication algorithm that comes much closer to the limits.

The starting idea here is to (temporarily) give up on integer multiplication and look at the multiplication of polynomials. Karatsuba can then be seen as looking at the identity

$$(at + b)(ct + d) = act^2 + (ac + bd - (a - b)(c - d))t + bd$$

There will obviously (?) be analogous identities for higher degree polynomials that might lead to yet more complicated fast multiplication algorithms. However a slightly different interpretation will turn out to be productive. Consider $a(t) = at + b, c(t) = ct + d$ then the three sub-multiplications we do in Karatsuba can be viewed as coming from the products of $a(t)$ and $c(t)$ at $t = 0, t = -1$ and $t = \infty$.

This insight leads to the suggestion for multiplying two polynomials (which I will now generalise to be of such a degree that their product will be of degree $n - 1$): select n different evaluation points z_0 to z_{n-1}. Evaluate each polynomial at each z_j and form the product of the values so found. Interpolate to find the coefficients of the product polynomial, which is known to be of degree $n - 1$. Because we know its value at n (distinct) points the product polynomial is uniquely defined. This scheme with a quadratic product and especially simple evaluation points leads to Karatsuba. The issue now is to see how it generalises!

Here I will refer you to Cormen et al who have a chapter on polynomial multiplication that my lectures will follow reasonably closely. The presentation in Knuth is more directly concerned with the application to integer multiplication. I will not cover the Cormen discussion of efficient FFT implementation (although it is interesting and useful) but I am confident that nothing I could write in these notes could match the presentation in these two classic textbooks. Note in particular that exercise 32.2.6 in Cormen et al gives the clue to what is needed for the integer multiplication case. You will see in these books that the apparently painful processes of evaluation and interpolation can be performed in $O(n \log n)$ steps. The multiplication after the polynomials have been evaluated may itself involve long arithmetic and this leads to a complete method that has cost $O(n \log n \log \log n)$.

Given fast algorithms for integer multiplication what are the consequences for other operations, such as division and the calculation of high precision square roots? It turns out that these are all related. In particular if we now just suppose that there is some function $M(n)$ that grows at least linearly with n and gives the cost of multiplication then we can derive time bounds on other operations.

Observe that if $y = 1/x$ then we can defined $f(y) = x - 1/y$, $f'(y) = 1/y^2$ and a Newton iteration for solving $f(y) = 0$ is $y_{n+1} = y_n(2 - xy_n)$. Each step of this iteration takes two multiplications (plus a subtraction). Because Newton's method exhibits second-order convergence it will be possible to obtain y correct to n bits in around $\log n$ iterations. This calculation is of course not one using integers, but the cost of the n-bit precision floating point arithmetic is just about the same as the cost of n-bit integer working. Now note that in the early stages of Newton's iteration the results are known only to have a small number of correct bits. Using full n-bit precision is overkill. One can use n-bit arithmetic for the final iteration, $n/2$ for the penultimate one, $n/4$ before that and so on. The amazing end result is that the total cost of forming a high precision reciprocal is bounded by (fairly small) constant time $M(n)$. Now obviously one can compute a/b as $(1/b)a$ so division is at worst a constant factor more expensive that multiplication.

Could division in fact be cheaper? No. Consider $1 + \frac{2}{\frac{1}{a-1} - \frac{1}{a+1}}$ which involves addition, subtraction and reciprocal operations. It should simplify to just a^2. In consequence one can square a number in time that is at worst a constant multiple of that needed to compute a reciprocal. Now $((a + b)^2 - (a - b)^2)/4 = ab$ and so if you can compute squares fast (and divide by constants such as four) you can do general multiplication. So multiplication can be done in time bounded by a constant compared with the cost of division. The two problems have been shown to be inextricably linked in computational complexity. This was done by showing how to reduce an instance of one to cases of the other. This technique is generally very important.

At this stage I include a further example to remind one that the search for the very best methods of solving problems may be hard. Consider the (pseudo-)code[2]:

```
a = 1;
b = 1/sqrt(2);
u= 1/4;
x = 1;
pn = 4;
for (;;)
{  p = pn;
   y = a;
   a = (a+b)/2;
   b = sqrt(y*b);
   u = u-x*(a-y)*(a-y);
   x = 2*x;
   pn = a*a/u;
   System.out.println("pn=" + pn);
   if (pn >= p) break;
}
System.out.println(p);
```

When I run this code the results I get are

```
pn=3.1876726427121086272019299705253692326510535718594
pn=3.1416802932976532939180704245600093827957194388154
pn=3.1415926538954464960029147588180434861088792372613
pn=3.1415926535897932384663606027066313217577024113424
pn=3.1415926535897932384626433832795028841971699491647
pn=3.1415926535897932384626433832795028841971693993751
pn=3.1415926535897932384626433832795028841971693993751
3.1415926535897932384626433832795028841971693993751
```

The curious iteration in fact makes it possible to compute π in time $O(\log n M(n))$. Its justification requires an understanding of elliptic integrals and extends to providing similarly fast ways of computing almost all of the elementary functions. It maybe helps stress the view that just because you have thought hard and can not find a better way of doing things does not mean there may not be one!

[2]Those who want to try this as an exercise can investigate the Java `BigDecimal` class that would make it easy to perform all of the long arithmetic.

3 Difficult

In this section I will explore the boundary between tasks that are practical and realistic to solve and those that are not. It turns out that the boundary is remarkably finely drawn — one can have two closely related problems one of which turns out to be easy to solve and the other hard. It also turns out that there is a rich and entertaining structure and theory associated with problems that lie close to the edge, and some famous unresolved questions.

This section contains material covered by Cormen et al, but my coverage will be closer to that given by Sudkamp. Very many other books cover the same ideas, but I think these two are among the clearer and their value for supporting other courses make them good buys.

A vital simplification is needed to make this part of the course work well. Although it may appear unreasonably coarse, it has in fact been found to be a useful predictor of practical reality. The simplification is to suppose that any calculation whose computing cost can be bounded by some polynomial in the problem size is a feasible one. Anything worse (and the easiest and most common growth rates that are worse than polynomial are the exponential ones such as $n2^n$ and $n!$) are considered infeasible. Note that this convention would view $10^{70}n^{1000}$ as "feasible", while declaring that $1.00001^{n/1000000}$ represented an exponential form of growth and hence unreasonable costs. One can take some compensation from that fact that even in the reasonably extreme cases given for sufficiently large values of n the exponential growth rate will exceed the polynomial one.

The investigation of polynomial vs. worse computing costs is a curious mixture. There are a number of ways in which it is essential to be carefully precise about what is being discussed. These include the following

Decision Problems: I will generally look at problems that yield just yes/no answers. On consequence of this (which I will not explore much here, but which is very much part of the language used in the field) is that once can consider a problem as defining a language. Each instance that will lead to a "yes" represents a sentence in the language. In line with Regular Languages and Automata theory one replaces the concept "solve a problem or perform a computation" by one of "accept a language";

Class of problem: All measurements are in terms of growth rates of cost across a family of problems. The analysis given here is not at all applicable to individual instances of problems or to problem sets that are finite. Where there is a parameter n that characterises the "size" of a problem I have to be able to think of what happens as n tends to infinity.

Decidable: Of course to be able to consider costs I had better restrict attention to decidable problems! And of course this means that I am only going to like algorithms that are guaranteed to terminate.

Worst case: Even if **most** instances of a problem are easy to solve, it will be the costs of solving the minority of worst-case examples that will be the focus of my attention. Thus in particular if I show that some problem has a worst case solution cost that is $O(2^n)$ I need to remember that the big-O notation is just giving an upper bound on costs and not necessarily a good prediction.

Size measurement: I need to formalise what I mean by the "size" (n) of a problem. Closely related to this is the issue of how particular instances of a problem will be represented as input to algorithms. In some cases alternative encoding of input can have a huge effect on the cost predictions for algorithms.

What computer: Perhaps some computers are faster than others. To avoid ambiguity here I will (when put under pressure on this matter) suppose that all computation is to be performed using a Turing machine, and that the input date specifying a problem instance is placed on the machine's tape. The number of tape cells used to store the input will be the measure n I use. How much difference does it make that I specify a Turing machine here? How on earth can I justify use of such a clumsy abstraction when talking about the need for fast and feasible calculations?

Let me start with the last of the above remarks. I will quote (but not work through a formal proof) a result that if an "ordinary" computer with directly addressable memory can run a program in time N than a Turing machine can simulate the behaviour of that ordinary computer performing the calculation, and complete its work in time $O(N^2)$. This result is enough to show that polynomial time calculation on ordinary and Turing computers are equivalent. There are some unexpected delicacies if you try to use more obvious abstractions of ordinary computation. For instance if you allow your computer simple integer arithmetic and count each arithmetic step as having unit cost then when you seek minimal costs and best algorithms unwanted tricks that use arithmetic and boolean operations on very very big integers tend to crop up in unwanted ways. Settling on Turing machines as the basis for formal discussion keeps everything safe, and the results that emerge appear to match up well with intuition. If a problem can be solved in polynomial time using an ordinary TM the problem is said to be in the class $\mathcal{P}$.

The big trick in this section is the construction of a technique that can be used to prove that a large number of interesting tasks are (probably) hard to solve. Note that one can show that a task can be solved cheaply by presenting a concrete algorithm that solves it and that is efficient. It is generally much harder to show that something is expensive. But I will be able to do just that not just for a single problem but for quite a few. The idea will be to show that **if** you could solve the problem that is under consideration easily (in this context the term "easily" means "with a computing time bounded by some polynomial") then you could solve some reference difficult problem "easily" as well. So we have two main steps to go through: the selection of a reference problem that we believe is hard, and the proof of a link between this reference problem and the one we want to study.

There are many ways in which this scenario has been applied, but the first and most critical one centres around the idea of non-deterministic Turing machines! Call these NDTMs. These are to ordinary Turning machines what non-deterministic acceptors are to regular (deterministic) finite automata. And as is true in the finite case it is possible to construct a deterministic TM that models the behaviour of a given non-deterministic one, and hence non-determinism does not allow us to solve any new problems. But it might give us a speed up. Define a non-deterministic solution as arising if **some** way in which the NDTM could calculate the desired result, and the cost of the calculation is just the number of steps taken in this successful path. An NDTM is not allowed to halt unless it can be certain that the output it has produced is correct (otherwise one could devise one that just wrote non-deterministic symbols to the output tape and halted - and then there would certainly be a possibility that it would produce the correct answer). An informal characterisation of calculations that a NDTM can do are those where an answer, if given, can be checked for correctness. Things that a NDTM can do fast are then ones where correct results can be recognised efficiently. An amazing number of interesting and worthwhile tasks happen to fit this pattern. Let me just start off by citing the problem of finding a route out of a maze — at each junction one can use non-determinism to decide which way to go, but when you finally escape the fact that you have reached the exit is utterly clear and visible. Problems that can be solved in polynomial time using a NDTM are known as $\mathcal{NP}$ problems. Remember that I am looking at families of problems so this is asymptotic behaviour, and that the polynomial bound on the cost must apply even in the worst case. The fact that problem is $\mathcal{NP}$ means that it is easy to give an algorithm based on some sort of exhaustive search through the possibilities that will solve the problem on an ordinary TM, but this algorithm will usually have exponential costs. Very extensive studies of huge numbers of $\mathcal{NP}$ problems have been made. Note that to say a problem is $\mathcal{NP}$ is in fact a comment on how easy it is. It can be solved in polynomial time given an NDTM. If in fact the problem can be solved in polynomial time on an ordinary TM then certainly an NDTM can solve it fast. The problem that arises is;

Are there any problems in $\mathcal{NP}$ that are not in $\mathcal{P}$?

Intuition, reasonableness and experience gives a clear impression that non-determinism in the sense discussed here does give very real extra power, and of course there must be problems in $\mathcal{NP}$ but not $\mathcal{P}$. But so far nobody has been able to prove this! Equally nobody has disproved it, and all the experts in the field believe it. Even with this problem left as open it is possible to direct attention to some of the hardest problems in $\mathcal{NP}$, ones that seem to make essential use of the non-determinism. These are the ones that will fail to be in $\mathcal{P}$ if any $\mathcal{NP}$ problems are. They are known as $\mathcal{NP}$ complete. A problem is $\mathcal{NP}$ complete if showing that that problem was in $\mathcal{P}$ would be enough to prove that $\mathcal{P} = \mathcal{NP}$. There is a more concrete was of expressing this definition. Suppose you have a problem Q that is $\mathcal{NP}$ complete, and you have another problem R that is $\mathcal{NP}$ (R can be any $\mathcal{NP}$ problem, not just limited to the $\mathcal{NP}$ ones). Then the availability of a deterministic polynomial time algorithm to solve Q would allow you to construct one for R. And that will be true for **any** R you care to select. This seems a convincing way of expressing the idea that Q was as hard as an $\mathcal{NP}$ problem can get. At first it seems improbable that there could be any $\mathcal{NP}$ complete problems!

I will sketch a direct proof that a problem called the "Boolean Satisfiability Problem 3-SAT" is $\mathcal{NP}$ complete. The problem 3-SAT is a somewhat abstract-looking one, but still has potential applications in hardware design and verification. An instance of it is a boolean formula in conjunctive normal form. This will be made out of a number of terms all **and**ed together. Each term will be made out just three items linked by **or** operators, and the items are either variables or negations of variables. A solution to a 3-SAT problem is a way of setting values for all the variables so that the entire expression evaluates to **true**. An NDTM can non-deterministically set values to all variables and then (easily) verify that the formula evaluates properly. A deterministic machine can obviously solve the problem by trying all combinations of values of the variables, but that may have exponential cost. Allowing more complicated forms of boolean formula could only make the problem harder, so the limit of CNF and three items per term are acceptable. I assert Cook's Theorem, that 3-SAT is $\mathcal{NP}$ complete.

To do this, consider an arbitrary $\mathcal{NP}$ problem of size n from class R. Because it is $\mathcal{NP}$ it can be solved on an NDTM in time $p(n)$ where the function p is bounded by some polynomial. If a Turning machine only runs for k steps it can only possibly move its head over k tape cells, so as well as only needing time $p(n)$ the solution of the problem only needs $p(n)$ cells of Turing Tape. In the lectures I will show how this makes it possible to construct a ridiculous large boolean formula that characterises the behaviour of the NDTM while solving the problem. I will also show that the size of this formula is bounded by a polynomial function of n. Now suppose that we could solve every instance of 3-SAT in polynomial time on a deterministic machine. This would now allow us to solve the instance of 3-SAT corresponding to the calculation involving R. With remarkably little effort we could then read of the solution to our original arbitrary problem. The effect overall would have been that the $\mathcal{P}$ solution to 3-SAT had allowed me to solve the arbitrary $\mathcal{NP}$ problem R in polynomial time. That would amount to showing that R was in $\mathcal{P}$ and so all of $\mathcal{NP}$ was in $\mathcal{P}$.

I will restate a few definitions:

NDTM: A Turing Machine which instead of having a transition function has a transition relation, ie given a symbol and a state it may progress into one of several different successor states;

A problem (instance) of size n: Given an understanding of what problem class and which NDTM is to solve it, the particular problem instance can be characterised by n symbols of initial data on the TM's tape;

Solving a problem in time t: An NDTM solves a problem in time t if (supposing the answer to this instance of the problem is "yes") there is at least one way in which the TM can behave where it stops after no more than t steps. If the answer to the problem is "no" then no possible computation by the Turing machine may stop, and certainly none may stop within t steps. Note that if the TM used happened to be deterministic we could run it for t steps and then give a definite answer yes or no. If non-deterministic we could run all possible ways it might compute for t steps and if one of these terminated report yes, and if none did report no — but this could be exponentially expensive;

Equivalent: Models of computation and problem encoding schemes are considered equivalent if each can simulate or be converted to the other and the overhead or cost of doing so is bounded by some polynomial in the size of the problem being solved;

Feasible: Solving a problem is considered feasible if there is a polynomial function $p(n)$ such that any instance of the problem of size n can be solved in time $p(n)$;

$\mathcal{P}$: The set of all problems that can be solved in polynomial time using a deterministic computer;

$\mathcal{NP}$: The set of all problems that can be solved in polynomial time if a non-deterministic computer is used. Very obviously $\mathcal{P} \subseteq \mathcal{NP}$, and it seems very plausible indeed that this is a strict inclusion, but nobody has proved that;

$\mathcal{NP}$-complete: A problem X is $\mathcal{NP}$-complete if $X \in \mathcal{P}$ would imply $\mathcal{P} = \mathcal{NP}$. The lectures spelt this concise explanation out in more wordy ways.

To prove that n-SAT is $\mathcal{NP}$-complete I first have to show that it is $\mathcal{NP}$ and then that it is complete. The first of these is just a demonstration that I can design an NDTM that will solve n-SAT sufficiently efficiently. Basically the NDTM will use its non-determinism to "choose" values to give to all the variables in the instance of n-SAT and then just check that the resulting formula evaluates to "true". Writing a program to do this for a Turing Machine would be messy but it is "obvious" that it could be done and that the program would run in a time bounded by a low degree polynomial (probably quadratic) in the length of the input. Most people would not view it is important to be any more formal than this since this is really pretty clear cut, but anybody who did could just design and code the details of the TM and then analyse its costs.

The interesting part is the bit about "complete". This will prove that if n-SAT is in $\mathcal{P}$ then all other problems in $\mathcal{NP}$ are too. Take an arbitrary class of problem from $\mathcal{NP}$, and select from it an example instance of size n. Because the problem is in $\mathcal{NP}$ we can produce an NDTM (say with m states) that will solve the problem within some definite polynomial time bound $T = p(n)$. We will consider the sequence of steps that correspond to the successful computation made by this NDTM. Since it only takes T steps it can only possibly inspect that much of the tape.

The entire behaviour of any successful TM calculation can now be described by showing what the TM's state is at each time step between 0 and $p(n)$, where its read-write head is on the tape and what symbol is present in each tape cell. To encode this as a boolean formula I introduce a collection of boolean variables:

$q_{k,t}$: Here I introduce mT variables. Remember that the TM had m possible states and it was running for T time steps. For any given value of t I will want just one of these variables to by true, and that will tell me what state the TM is in at that time;

$h_{i,t}$: The variable $h_{i,t}$ will be true if the TM's head is at position i at time t. Because both the length of the tape and the total time-steps taken are just T I have T^2 variables here;

$s_{i,t,x}$: If the TM's alphabet is Σ then I have ΣT^2 variables here, where if one of them is true it indicates that tape position i holds symbol x at time t.

A proper assignment of truth-values to these boolean variables would show me exactly how the TM behaved, and as the size of the problem the TM is solving grows the number of variables that have to be used grows proportional to T^2, is $p(n)^2$, still a polynomial in terms of the problem size. Next I need to write out a boolean formula in terms of these variables. Since I will want to construct one if the form n-SAT its top level will be a lot of terms, all of which get anded together.

I will describe these terms in groups before I look what happens when I combine them all. Before doing so I will observe that I can make life easier for myself by writing terms in the form

$$p\&q\&r \Rightarrow (x|y|z)$$

since the implication can be re-written later as the more basic form

$$\neg p|\neg q|\neg r|x|y|z)$$

So here are the terms I will put into the final instance of n-SAT:

Start in state 0: $(q_{0,0})$;

End in the halting state (state h, say): $(q_{h,T})$;

At each time I am only in one state: $(q_{k,t} \Rightarrow \neg q_{k',t})$. I have to list this for all pairs of (distinct) state k and k' and all times t, so there are m^2T terms of this form to list (and **and** together);

Head position unambiguous: $(h_{i,t} \Rightarrow \neg h_{i',t})$. This represents T^2 terms in the same sort of way;

Only one symbol at each tape position: $(s_{i,t,x} \Rightarrow \neg s_{i,t,x'})$. I have $|\Sigma|T^2$ terms here, and again just **and** them all together;

At time=0 the tape contains $[a, b, c, d \ldots]$: $s_{0,0,a}\&s_{1,0,b}\&s_{2,0,c}\&s_{3,0,d}\ldots$ is a collection of terms to put into my eventual formula that enforce the correct initial tape contents, and which only add a very modest bulk;

Symbols do not change away from the head: $s_{i,t,x}\&\neg h_{i,t} \Rightarrow s_{i,t+1,x}$;

The TM state transitions are correct: $q_{k,t}\&h_{i,t}\&s_{i,t,x} \Rightarrow (q_{k',t+1}|q_{k'',t+1})$. This is the clever rule. The states k' and k'' represent two possible successor states, and this makes the TM I am describing a non-deterministic one. It is necessary to list $m|\Sigma|T^2$ implications of this form to cover all the possibilities;

the TM writes proper symbols back to the tape: $q_{k,t}\&h_{i,t}\&s_{i,t,x} \Rightarrow s_{i,t+1,x'}$. This is very similar to the previous rule except that without loss of generality I can make the non-determinism show up only in state transitions not in the symbols used. once again the number of terms that have to be generated is proportional to T^2;

The TM moves left and right as it should: $q_{k,t}\&h_{i,t}\&s_{i,t,x} \Rightarrow h_{i\pm1,t+1}$. Obviously the $+$ or $-$ case will be used depending on whether the TM moves its head left or right in the given circumstances.

If all the above are combined together with **and** operators the result is a formula in the format that n-SAT expects such that the formula can (obviously) only be satisfied if and only if the TM has a terminating computation. As the length of the TM calculation, T, grows the length of this formula grows to have a number of symbols in it that is proportional to T^2. Pedants will explain that we need $\log(T^2)$ times that many characters if the variables are to be named $v0000$, $v0001$, $v0002$ and things like that so that a fixed alphabet is used, but this is a pretty trivial expansion. It is furthermore important to know that the instance of n-SAT could be constructed in polynomial time from a description of the TM. The simplicity of the rules I used to generate it make me feel that I can say that that is pretty obvious too.

Using all that has gone before I can now assert that n-SAT is $\mathcal{NP}$ complete. Suppose I have an arbitrary $\mathcal{NP}$ problem and that n-SAT is in $\mathcal{P}$. Design an NDTM to solve the arbitrary problem. Use the rules given above to construct an instance of n-SAT that describes how the NDTM runs. Because I have suppose n-SAT is in $\mathcal{P}$ I can now solve this in polynomial time — a polynomial in the size of the boolean formula. But this is still a polynomial function of the size of the original arbitrary problem, and from the solution to n-SAT I can read off a solution to the original task. Thus I have just solved the arbitrary $\mathcal{NP}$ problem in polynomial time on a deterministic computer, and hence if n-SAT $\in \mathcal{P}$ I would have shown that $\mathcal{P} = \mathcal{NP}$.

This was historically the first problem proved to be $\mathcal{NP}$ complete, and still about the easiest one to deal with directly.

I next go on to discuss some of the many other problems that have been proved to be $\mathcal{NP}$ complete, and in particular show how to reduce the Hamiltonian Circuit (which I will define) problem to 3-SAT and hence prove it $\mathcal{NP}$ complete. I also cover the Travelling Salesman Problem (TSP), searching for k-cliques in graphs, the integer knapsack problem and others. You can see eg Cormen at al for the Clique problem and Sudkamp for the Hamiltonian circuit. You should note that for all the results proved there will be multiple (and equally valid) proof strategies, and so the selection of particular ones is based on a judgement about which seem easiest to follow.

Once a problem has been proved $\mathcal{NP}$ complete it is tempting to give up on it. But there are more things to do. Firstly, restricted forms of some $\mathcal{NP}$ complete problems are in $\mathcal{P}$. Secondly many $\mathcal{NP}$ problems relate to forms of optimisation where a search for a "best" solution is called for. Sometimes (but not always) slackening off the constraint and seeking a "reasonable" rather than optimal solution can lead to solutions in $\mathcal{P}$. Finally the analysis here is in terms of worst case costs, and amazingly often for $\mathcal{NP}$ complete problems there can be algorithms that are usually (in a rather informal sense!) efficient or which usually get the correct answer. I will survey these ways of facing up to the practical problem that

$\mathcal{NP}$ completeness represents.

4 A non-elementary problem

The final section of this course covers a problem that is decidable, and indeed most of you could write a program that would solve it. However it has the property that even the best possible program (presumably using techniques much more devious than any you would code up) will have a worst case running time that is worse than 2^n, worse than 2^{2^n}, and indeed worse than *any* such formula with any fixed number of levels of exponentiation. The problem uses "Extended Regular Expressions" and given one of these asks if the language that it defines has empty complement. This is a short title for the problem but it calls for a little more explanation. I should note that the main place I know where this result is documented clearly is Aho Hopcroft and Ullman's book on the Design and Analysis of Algorithms, but that I hope that the notes I include here (together with the lectures) are enough to support the level of understanding I expect from you here.

4.1 What is an extended RE?

Ordinary Regular Expressions are built up starting from some base cases that include a denotation for the empty language, for the language that consists only of the empty string, and from symbols that stand for each letter of the alphabet. Whenever a regular expression is written there ought to be at least an implicit understanding of what the alphabet Σ is within which we are working. The three combining rules for REs are

Concatenation: if `A` and `B` are regular expressions then `AB` stands for one that matches an `A` followed by a `B`;

Alternation: `A|B` matches anything that matches either `A` or `B`;

Arbitrary repetition: $\mathtt{A}^*$ is much the same as $\epsilon\,|\,\mathtt{A}\,|\,\mathtt{AA}\,|\,\mathtt{AAA}\,|\ldots$, ie it matches strings that can (somehow) be decomposed into zero or more segments each of which matches `A`.

It is well known that a Regular Expression defines a Regular Language, and that Regular Languages form a class closed under intersection and complement. Thus if one were to extent the definition of regular expressions to permit

Intersection: if `A` and `B` are regular expressions then `A&B` stands for one that matches an `A` and also matches `B`;

Negation: $\neg$`A` matches and string over the alphabet Σ that `A` does not.

then this does not alter the class of languages that can be described. The most it can do is to give up neater and easier ways of describing some of these. For instance over the alphabet that includes all letters from `a` to `e` the extended regular expression $\neg$`a` represents all strings *except* for the one that is made up of the single letter `a`. To write this without a negation mark is possible, but actually quite messy. About the best I can do easily represents it as two alternatives, the first is all strings that do not contain `a` at all, and the other is the all strings of length at least 2.

```
(b|c|d|e)* | (a|b|c|d|e)(a|b|c|d|e)(a|b|c|d|e)*
```

Perhaps you can find something shorter, but I hope I have made the point that the notation $\neg$`a` may be convenient.

Another well known result about regular expression is that given any regular expression (and this will include extended ones) there is a systematic way of constructing a deterministic finite automaton that will accept the associates language. This result is part of Kleene's theorem.

4.2 Emptiness of complement

Now I introduce the *emptiness of complement problem*. For a regular expression `R` this asks if the language defined by $\neg$`R` is empty. Empty languages may seem a silly thing to consider, and looking for an empty complement rather than an empty language may also appear artificial. Let me at least respond to the first. If `A` and `B` are *big* regular expressions we might be interested in trying to decide if they are equivalent in the sense that they define the same language. Is it possible to do this? Well consider `A&B` | $\neg$`A&`$\neg$`B`: if the two expressions denote the same language to start with this is an extended regular expression with an empty complement. So apart from theoretical interest this problem, if solved, might help us check REs for equality.

The emptiness of complement problem is decidable. Take the target RE and construct from it a finite automaton `M`. This process is described in the proof of Kleene's theorem and is mechanical — it does not give us any undecidability worries. If you attended and understood either a Part IA Regular Languages course or a Diploma course on Mathematics for Computation Theory you will have seen this and could possibly see that you could write a program to perform the conversion. Otherwise earlier chapters of Sudkamp's book may help you. Here I just need to quote the result not re-prove it. Suppose that `M` has n states, then if `M` accepts a language with empty complement there must be some string of length at most n that it rejects. This result is based on the usual analysis of the fact that in any computation that accepts (or rejects) a string of length greater than n the machine must have repeated a state. Thus an utterly crude way to check things is to generate all the strings up up to n symbols over our alphabet and check whether `M` accepts them all. This is intolerably tedious if we are practical people, but enough to establish that there is a systematic and algorithmic method of resolving our problem, ie it is decidable.

4.3 Introduction to proof of hardness

I will now sketch a demonstration that the emptiness of complement problem for extended regular expressions is astonishingly expensive to solve. A proper presentation can be found in Aho Hopcroft and Ullman's book, so my attempt here will be to get across key ideas. What I give here will not be a complete proof, and every so often I will rely on plausible (and true) sub-results.

The idea behind this is going to be based on calculations performed by Turning Machines when they have a limited length of tape to work with. Note that this bears a distinct resemblance to the starting point we had for $\mathcal{NP}$-completeness analysis via 2-SAT. The insight needed to start us off is that if we can describe the behaviour of Turing machines that are restricted to using up to k symbols on their tape then telling if those machines are every liable to terminate will necessarily take an amount of time that grows as k does. I guess I need to say that the halting problem for unbounded Turing Machines is undecidable, but if you have a TM which has N internal states, uses an alphabet of size m and limits its tape to length k then there are only Ns^k configurations it can possibly be in and analysing its potential behaviours just amounts to looking at a finite automaton of that (large) size. Note that I am not proving that the analysis of tape-bounded TMs is difficult here, I am supposing that somebody else has done that for me, and anyway I do not expect you to find it an astonishing result. I am also not too worried about *exactly* how difficult it is: all I need is that if you give a TM a longer tape then it gets harder to analyse its potential behaviours.

Now a TM with tape limited to length k is really a finite automaton, so Kleene's theorem tells us that its behaviour can be described using regular expressions. What is (I hope) astonishing is that I will be able to show that this horrible-sounding conversion is in fact quite easy to do, and does not involve all the general mechanism you saw in the full proof of Kleene's theorem. Furthermore it I use extended regular expressions the extended-RE I get will be quite compact. This way of starting from a TM and ending with an extended RE is the key step. In style it is just like the reductions done when proving results about $\mathcal{NP}$ problems but here it will provide a proof (by reduction) that the emptiness of complement problem is as hard as deciding the termination properties of a tM with a *very* long tape.

OK, so my TM has N states and an alphabet of size m. Its tape is k long.

For my RE I will want a new alphabet that consists of N symbols that stand for TM states (I will write these as Q_i), together with m symbols that are just from the TMs alphabet (I will use a_i as such a symbol) and a new symbol `$` that I will use as a delimiter. I will call the size of this enlarged alphabet M. The TM will have a transition function that shows what it has to do for all the Nm possible combinations of state and symbol.

Now I will define a *trace* of the behaviour of the TM. It is a string with $-signs every k characters, and the text between each pair of stars writes out the state of the TM tape at one time step. The position of the read-write head is indicated by putting a Q_i just to the left of the symbol about to be read. Successive blocks of the trace represent the state of the TM at consecutive time points. The first block of trace is its initial state, and in the final block the state Q is a halting one.

It is probably neatest to display traces on several lines, with a line- break inserted before each $. Then corresponding positions on the tape of the TM will align vertically. The following trace is for a silly TM that stars with a tape full of `b` symbols and scans rightwards turning them all into `a`. When it detects the end of its tape the read-write head moves one space left and the machine enters its halting state, which I have written as `H`.

```
$Qbbbbb
$aQbbbb
$aaQbbb
$aaaQbb
$aaaaQb
$aaaaaQ
$aaaaHa
$
```

If you are upset about the idea of "detecting the end of its tape" do not be. You can think of the $ as a special symbol pre-written where the tape is supposed to end. If you want to work out an example for yourselves I would suggest that you design a TM that works on an alphabet that is `0`, verb.1. and a few control symbols, and that counts in binary through all the k-bit numbers before it stops. Doing so will count as revision for the Computation Theory course more than work on this one.

I intend to show how to create an extended RE that will match only those strings that are proper traces for some given TM.

For the bulk of this explanation I am going to dodge one critical issue, which is that of ensuring that items in a string are spaced a fixed (and typically large) distance apart. My temporary fudge will be to use a notation x^k to stand for k copies of the symbol x. I will also use a wild-card "?" that is really an abbreviation for $(a|b|\ldots)$ alternating over all my alphabet. There is a risk that these short-hands will let me write things down unduly compactly, so I will need to show (later on) that once I have a regular expression using them I can expand it out to fit in with just the rules of extended regular expressions without undue expansion.

4.4 Constructing an extended RE for a trace

Because I am working with extended regular expressions I can either try to build up a single expression that will match all valid traces, or I can build up one that will detect all **invalid** traces (and if necessary take a complement at the end to get the one I actually need). The two ways of working are pretty well equivalent: in one the little fragments that I generate will need to be linked together with *and* operations and in the other I will need a lot of *or* symbols. In the lectures I will probably present the construction as of an extended RE that would match all valid traces. This choice might be justifiable because it involves making a positive statement about a trace, and that is perhaps easier to understand than a negative one. Here I will write things out as descriptions of the ways in which a trace can fail to be valid. The advantage of this is that it is perhaps easier to verify that all possible failure modes have been described than it is to ensure that the RE components fully enforce proper behaviour. But I stress that the two presentations differ only by the insertion of a few "¬" signs and maybe use of DeMorgan's rule. For Extended Regular Expressions the extra "¬" operators are not an issue. Looking at semi-extended regular expressions (where you permit *and* but not *not*) would mean you had to be a bit more careful, and Aho Hopcroft and Ullman do that.

To build up my whole extended RE I will set up a sequence of separate sub-expressions each of which detects *some* way in which a trace could not match the Turing machine's behaviour, and then I will just combine all these with "—" operators to make a bigger RE that can detect all possible discrepancies. The complement of the language defined by this big extended RE will clearly consist just of traces of proper terminating computations that the TM can perform. Thus the complement will be non-empty only if the Turing machine has a terminating computation. Please remember that this is *not* the same as the Halting Problem because here I am limiting the TMs to use only a tape of length k.

I will start off with the easy ones:

Must have dollars after every k characters *I give here a rather simple RE that describes valid configurations on the tape. Take its complement to get all the invalid ones:*

$$\$((\neg\$)^k\$)^*$$

A dollar sign then blocks each of which is a chunk of k symbols that are not dollars and then another dollar.

State Q_0 at start of tape to begin *Again I will describe things in a positive way here and take the complement of this before combining with top-level "—" operators*

$$\$Q_0?^*$$

Initial symbols on TM tape *In the trace it is important that there be only a single symbol from the set Q in each display of the TMs state. For all except the first block this will be enforced by the rules for the TM transition, but something is needed to indicate that to start with the tape contains only symbols from the TM alphabet. I will allow my RE to describe all possible TM calculations for this machine, and hence will permit the machine to start with arbitrary tape contents. In fact this rule is just an extension of the previous one:*

$$\$Q_0(a_0|a_1\ldots)^*\$?^*$$

The only things I allow after the two initial symbols \$ and Q_0 are things from the TM's alphabet, all the way until I find the next \$.

Symbols on the tape do not change arbitrarily

$$?^*(\neg(Q_0|Q_1\ldots))a?^k\neg a?^*$$

This is my first case where it has been nicest to describe what must not happen rather than what must. The $?^*$ at the start and end then just permit recognition of the bad thing anywhere within the whole trace. Now the idea is that this one sub-expression I have written should be thought of as written out lots of times, one for each possible Turing Machine symbol a. It then says (actually rather directly) that if you see a symbol a in the tape and it is not preceded by a state (ie if the Turing machine head is not reading it at the moment) and if the symbol that arises just k symbols later (ie at the corresponding position in the trace for the next time step) differs then we do not have a good trace. That feels like a mouth-full, and it does mean that I have just listed m^3 chunks of regular expression. But m is a constant so that does not worry me.

The TM's transition rule applies *The transition rule will show have explanations of what to do for any combination of Turing Machine symbol and state. It documents what symbol is written back to the tape, whether the read-write head moves left or right and what the new TM state should be. Again it turns out to be best to make a list of all ways that this could be*

[3] Remember m was the size of my alphabet, and so is fixed.

violated. The expected pattern when the TM moves left will look something like

$$aQb\dots Q'ab'$$

where the dots must represent exactly $k-1$ symbols. The idea is that the TM is in state Q and is reading symbol b. To its left is a symbol a. In the next block of the tape the TM head has moved left so it will next read a, and it has changed the b into b' and set its new state as Q'. To write out all ways in which this might be violated involves listing expressions for all possible values of a, b and Q, but that still is only m^2N cases.

$$?^*aQb?^{k-1}((\neg Q')|(Q\neg a)|(Qa\neg b'))?^*$$

Must have halting state Q_h in last block

$$?^*Q_h(\neg(Q_0|Q_1|\dots)^*\$$$

This says that a Q_h is present, and that what follows it is a string made up of anything except other states Q_i and then the end of the trace. Again negate it to get an RE to describe violations.

I think that the above document all the properties needed to describe a proper Turing Machine trace. One thing to note is that the REs will permit the TMs tape to start off in utterly any state, so so solving the emptiness of complement problem for it amounts to deciding if there is *any* initial data that could be presented to this particular TM that would cause it to halt. In the above presentation I have still fudged the issue of symbols being separated by a distance k, and I have also used a collection of other short-hand notations. In a few places I have used "…" and I need to count how much text that really expand to, and in places I have written down one bit of RE but then indicated that it should be replicated lots of times. I have also used a question mark to stand for an arbitrary symbol.

Let me deal with the last of these first. The RE I have written as $?^*$ can be expanded in one of two ways:

$$\neg\phi$$

or

$$(\$|Q_0|Q_1|\dots|a_0|a_1|\dots)^*$$

The advantage of the first is that it is very compact, and for RE experts who are happy with the empty language ϕ it seems good. The second presentation just lists all symbols in the RE's alphabet explicitly: it is perhaps easier to understand (if more bulky). For this course I do not mind which you use. Aho Hopcroft and Ullman consider *semi-extended* REs as well as extended ones. These just add an *and* operation, not a *not* one, and and so they would probably prefer the second version.

Now how bulky is the extended RE I have got by combining all the components listed above? For a TM with N states using an alphabet of size m the whole extended RE will consist of around m^2N components (mostly from the parts that ensure that the TM transitions are correctly followed). Even if I do not allow use of a $\neg$ operator within my sub-expressions and so have to use the lengthy replacement for $\neg\phi$ (and similarly for $\neg a$, $\neg b'$ etc) each of these components is only of length some small constant time (m+N). Thus apart from the remaining oddity about strings of length k I can have an extended RE of size that depends only on the size of a description of the TM, not on the length of the computation it will perform.

4.5 Rulers

The remaining challenge is to take an RE that has been written using the informal extra notation $?^k$ in it to specify that some symbols are separated by a distance k, and show how I can expand things to get a simpler (but somewhat larger) RE that describes the same language.

I will do this first using a mechanism that works well for small values of k, and will then show how to extend it. Suppose for instance we wanted to do this for $k = 6$. I first write down the obvious and easy RE

$$\bullet^*\spadesuit\bullet\bullet\bullet\bullet\bullet\spadesuit\bullet^*$$

[Note that I have used a pair of slightly odd symbols here to stress that I want something that will not clash with the symbols in my original RE]. This clearly describes any string that has two $\spadesuit$ symbols marking the two ends of a sub-string of length 6. I will refer to such an RE as a *ruler*, in this case of length 6.

Now take my original RE, which works over some alphabet Σ and has somewhere in it the short-hand $?^k$. I will construct a new RE that works over a bigger alphabet, specifically $\Sigma\times\{\bullet\spadesuit\}$, ie ordered pairs from the original alphabet and the new one. Wherever in my original RE I had a symbol a ($a \in \Sigma$) I will now write our $([a,\bullet]|[a,\spadesuit])$. Doing this expand the bulk of the RE by a factor that is around the size of the marker alphabet $\{\bullet,\spadesuit\}$, but otherwise leaves the RE describing the same sets of strings. Similarly I rewrite my ruler replacing $\bullet$ by $([a_0,\bullet]|[a_1,\bullet],\ldots)$ and similarly for $\spadesuit$. The symbols a_i I am using here are supposed to be just a list of the symbols in the alphabet Σ. My description of the ruler has expanded by a factor of about $|\Sigma|$.

Now I can replace the previously troublesome segment $?^k$ with

$$[?,\spadesuit][?,\bullet]^*[?,\spadesuit]$$

which indicates that the first and last characters in the run must be paired with ♠ markers. The residual "?" marks can be expanded by just listing all the possible characters from Σ that could possibly arise. The effect is that the item $?^k$ has been replaced with something whose bulk is around $|\Sigma|$.

Finally I take the *and* of these two expressions. One preserves the behaviour of the whole on the first component of the new alphabet, while the second enforces the required separation of k positions.

If the original RE had an alphabet Σ and was of length l, while the ruler used an alphabet Σ' and had length l' then the new RE that I construct will have length around $l|\Sigma'| + l'|\Sigma|$.

I should add a technical note here: the above expansion removes a single instance of the idiom $?^k$ from an RE. One might fear that it was necessary to do that expansion all over again for each instance of the sub-text "$?^k$". At least in the application I have here this is not so and a single ruler can be used to enforce all of the measurement properties I need.

The key to making my overall results about simulating TMs interesting will be the construction of concise representations of very long rulers.

4.6 Long rulers

Suppose I have a ruler of length k. The first part of this section shows that I can use it to help me describe the behaviour of any Turing Machine that I like provided that that machine only ever looks at k cells on its tape. Now I know that in the first section I just described how to model Turing Machines with unspecified initial tape contents, but it is trivial to put in a bit of extra RE to force the TM to start with a blank tape. A trace of its behaviour then has length that is the length of the tape (ie k) times the number of steps the TM takes before it halts.

It is easy to produce a TM that uses an alphabet that consists of just "0", "1" and the end-of-tape marker $m which if started on a tape of length k that is initially filled with zeros counts up in binary until the tape eventually contains just ones, and then terminates. Indeed it is so easy that I will document it here! The machine I will describe has 3 states, and I will call then A, B and H. The behaviour and description of these states is as follows:

A: I will call this state "carry 1", and it is also the starting state of the machine. If it finds a symbol 0 it turns it into a 1, enters state B and moves left. If it finds a 1 it turns it into a 0, stays in state A and then moves right. If it reads a $ it switches to state H (and it does not matter very much which way it moves or what it does to the $;

B: This state may be interpreted as "scan back to the start of the number" and is entered whenever state A has finished incrementing the tape contents by 1.

Until it finds a \$ it just moves left, leaving the tape unchanged. On reading a \$ it moves right and enters state A;

H**:** This is the halting state.

This machine has three states and uses an alphabet of size 3 (including the \$ which I use to delimit the usable region of the tape). And given a tape of length k it clearly goes through 2^k occasions where the read-write head is at the start of the tape and it is about the increment the number. Well I suppose if one is being really pedantically careful here it may be necessary to wonder in detail whether the length of my ruler describes the length of the tape or the length of the tape plus or minus 1, and whether the length of the tape has to count the cell on which the \$ symbols are written. But such fine details only introduce a (small) constant factor possible difference and really do not matter, so I will continue to write just 2^k here. Now a trace of the behaviour of this machine (in the sense that I used the work *trace*) before) is distinctly longer than $k2^k$. All I will need to use here is that it is longer than 2^k so I do not even need to agonise about just how many steps it takes for the TM to perform each increment operation and then return to the start of the tape.

Using the construction given earlier I indicated that an extended RE could be produced for an N-state TM over an alphabet of size n that was around $Km^2N(m+N)$ (for some small constant K) symbols provided that at that stage I still permitted use of the informal notation "$?^k$". In this case $N = 3andm = 3$ so that is just $486K$ (regardless, at this stage, of the value of k). This RE will have as its alphabet one of size 6, ie $\{$0,1,\$,A,B,H$\}$.

Suppose my ruler of length k was itself an RE of length p and it used an alphabet of size q then I can put the ruler and my TM simulation together to get a single RE of size around $6p + 486Kq$. This bigger messier RE now uses an alphabet of size $6q$ (the product of the sizes of the two previous alphabets) but manages to describe strings that have a symbol H in them only after (rather more than) 2^k symbols.

Why have I done all of this? Well the new RE can now be adapted[4] to be a ruler whose length exceeds 2^k but its size is only a constant factor bigger than the size of the ruler we were using that was of length k. So now I can repeat the construction using this ruler of length 2^k to get a trace of length 2^{2^k} out of an RE that is again only a constant factor bigger than the one I already had. And I can keep on doing this over and over again. I can end up with a ruler of length

$$2^{2^{2^{\cdots^k}}}$$

[4]If one takes the trace described by such an RE and puts an additional H on the front then the two H symbols (there are exactly 2) serve in the way that the ♠ ones did before, while all the other symbols have to be treated as variations on •.

with any given height h of powering and the extended RE that describes it is of size that grows relatively[5] slowly with h.

The length I have for a ruler is at present expressed as a huge tower of exponentiation based on the parameter k that is the length of my initial ruler, while what I am interested in is expressing the length of my final ruler in terms of the size of the RE that describes it. I will omit the fine details of mathematical analysis to show that all is well here and that I can create rules whose length is astonishingly exponential in the size of the RE involved, and content myself with saying that what I have shown is that

1. I can produce a simple ruler of length k using around k symbols;
2. making a ruler that is exponentially bigger than an existing one only makes the RE for the ruler a modest factor bigger (because the alphabet that is to be used grows in size as well as the number of symbols written in the RE you may find that the bulk grows quadratically with the number of layers of exponentiation desired);
3. The difference in these growth rates means that you can just do the exponentiation step an extra time or so to ensure that the final ruler is huge compared with the RE that describes it rather than just being huge relative to k.

Mathematical pedants are welcome to follow up on the fine details!

4.7 Using a huge ruler

If I have a huge ruler I can readily describe the behaviour of Turing Machines only limited to that (huge) tape-length. And by selecting nasty enough such machines I can make it hard to tell if the TM can ever terminate with the difficulty growing with the tape length involved. I am not proving this here, just asserting it (again Aho Hopcroft and Ullman do have a proof, in case you are really keen, but to me it seems a jolly reasonable thing to expect to be true). Solving the emptiness of complement problem for my extended RE amounts to guaranteeing to be able to tell whether the TM I have modelled can ever terminate, so this problem is as hard as the length of the ruler I can describe, and so is at least as hard as something that is super-exponential with any number of powerings you care to mention. ie it is non-elementary.

[5] Almost **any** growth-rate would be modest relative to this!

5 Conclusion

These notes have provide an overview of the material covered in the course. Some of the detailed results and all of the exercises have been left to the (excellent) supporting textbooks. The unifying ideas that I hope come through include:

1. Seeking the very fastest (asymptotic) solution to a problem may lead to unexpectedly deep and complicated algorithms;

2. Finding lower bounds on the cost of solving problems is important but hard;

3. Relating the cost of pairs of problems is an important and general technique;

4. Computation by Turing machines provides a useful model for practical (ie computing-time analysis) as well as purely theoretical (ie just decidability) purposes;

5. Some tasks are easy, some hard and some *almost* impossible.

References

[1] Aho, Hopcroft, and Ullman. *The Design and Analysis of Algorithms*. Addison Wesley, 1976.

[2] Leiserson Cormen and Rivest. *An Introduction to Algorithms*. MIT and McGraw-Hill, 1990.

[3] Donald E. Kunth. *The Art of Computer Programming*, volume II. Addison Wesley, 3 edition, 1998.

[4] Thomas A Sudkamp. *Languages and Machines*. Addison Wesley, 1988.