Advanced Algorithms

A C Norman, Michaelmas Term 1995

Part II

1 Introduction

The Part IB course on Data Structures and Algorithms describes a wide range of practically valuable methods, many of which represent the fruits of a great deal of clever and ingenious design. But in general it concentrates on success stories, where a simple and obvious problem has a solution that appears to solve it completely, and that solution (while often unexpected or cunning) is fairly compact and comprehensible in retrospect. Part of the idea behind this Part II course is to show that the earlier course did not even do justice to the tip of the iceberg, and that the amount of work that has been done on algorithms and the amount yet to be completed is enormous. It also looks again at various procedures introduced elsewhere in the Tripos and asks the difficult question "Yes, but can you do yet better?" or "What are the detailed implications of the minor sub-tasks that were taken for granted in the big outline?". In several cases these lead to new and elaborate data structures, or forms of analysis that is harder than would have a place in a Part IB course.

This course is for Part II, and is new for 1995. There are various consequences of these two facts:

- I will not feel obliged to give the course at a level where **every** student can follow (and thus enjoy) it. In Part II students can choose from among a wide range of options and it is proper that some of these will not be at all suitable for all the class. I will, of course, feel embarrassed if I take such an aggressive stance in the course that the audience drops to single figures.

- These lecture notes will be less complete than ones I have tended to prepare for earlier years in the Tripos. Thus inspection of the textbooks will be called for.

- As with any new course there may be some teething problems: I have designed the syllabus to make it is safe as I reasonably could, but until I have given it once it is hard to know exactly how much time I will need to allocate to each topic. If I cover less (or indeed more!) than the "yellow book" and these printed notes suggest then I will adjust the examination questions accordingly.

- Supervisors may feel exposed if they offer to assist with this course. In fact I do not think I will be saying anything that ought to be a real problem in that respect, but a clinic at the end of the Term (Thursday 30th November, probably at 4pm, but further details will be circulated later) will provide opportunity for students (or indeed supervisors) to cross-question me or seek clarification.

- Part IB courses are given and attended as a matter of duty, while Part II ones are for enjoyment (on both sides). I think that the material I have assembled is utterly amazing and entertaining as well as being the sort of stuff that every well-educated Computer Scientist ought to know. Some is even directly applicable. It also shows that practically-motivated areas within the subject can be complicated and hard almost without limit.

- I will duck lots of formal proofs and fine details, taking the view that textbooks cover them much better than a lecturer can.

The course is structured in four two-lecture sections. It might be rational for students to assume that I will prepare one examination question on each section, discard the one that looks least well formed, and use the best random number source I have to choose which of the others I set. Potentially it would be possible to attend the course in modules, skipping one that felt especially unattractive. The topics to be considered are:

Binomial and Fibonacci Heaps: these grow out of a desire to solve the single-source shortest path problem (eg using Dijkstra's algorithm) on sparse graphs as fast as possible. Maybe the surprise is that Dijkstra's algorithm as documented at a IB level fails to think harder about this—but there again when you see how messy Fibonacci Heaps are you will understand!

Kolmogorov Complexity: Usually evaluations of the information content or complexity of things will be asymptotic, and arbitrary finite odd-cases will be quietly ignored. This part of the course provides some basis for considering a finite string such as "abababababab" to be less complex than one like "abbbaabaaabaaaa". This involves looking at various ideas about data compression, Turing machines, probabilities and information theory. Note very well that I will only give a simple introduction to a large and difficult subject area.

Probabilistic Algorithms: The relationship between true randomness and computer-generated pseudo-random numbers. The extent to which a truly random "oracle" might be a help in algorithm design, including a discussion of the fact that there are several different ways of producing precise models of what "successful random computation" might mean. One of the most important areas where probabilistic algorithms are commonplace: integer primality checking and factorisation. While I intend to duck most technical issues this borders on including a bit of computational number theory.

More Garbage Collection: The background challenge here will be to take a procedure which is easily implemented as one that takes place in large disruptive chunks and to derive schemes that support (near-) real-time processing.

I may discuss both "emphemeral" garbage collection and the issues behind true parallel garbage collection where one processor continuously allocates and uses memory while another rushes along in the background tidying up.

2 Binomial and Fibonacci Heaps

I will start with a review of Dijkstra's algorithm for the single-source shortest path problem. This is because it (at least on sparse graphs) helps to justify some of what follows. Suppose one has a directed graph with (positive integer) weights on the edges, and an initial special vertex v_0 called the "source". Then the algorithm will discover the length of the shortest path from v_0 to each of the other vertices in the graph. As a matter of convenience a special value ∞ is used to indicate that there is no path from the source to some particular vertex. Pedants can note that I only want finite values as weights on the edges, and that I do not want any arithmetic overflow in processing these distances to generate anything that can be confused with the formal ∞ that is used to label unreachable vertices. I will suppose that the graph has V vertices and E edges, and that it is represented in such a way that finding the edges that meet at a given node is cheap.

The algorithm uses a set Q that contains the collection of vertices not yet processed. This can start off consisting of all the vertices in the entire graph. v_0 will be marked with the value 0, while all other vertices start off labelled with ∞. The algorithm proceeds by iterating two steps until Q is empty:

1. Identify and remove the vertex with smallest label from Q. This will of course be v_0 on the first occasion. This step decreases the size of Q by one, and so the step is executed exactly V times. Call the extracted vertex u.

2. For each edge coming out of u and running to a vertex v, check if the label on u plus the weight on (u,v) is less than the current label recorded with v. If so reset the label on v to the new smaller value. Since the graph being worked on is a directed one it is easy to see that that step at worst updates weights on E occasions, since each vertex will take on the role u just once and at that stage all the edges coming out of it will be processed.

The overall cost thus involves V operations that identify the smallest item in a set and remove it, and E cases where a value in the set is decreased (note that values are only decreased, never increased). For a sparse graph it is productive to view Q as a priority queue. As will shortly be explained if this is implemented as an ordinary heap (as in heap-sort) each of the two operations has cost proportional to $\log(|Q|)$, and since Q contains a subset of the vertices this is roughly $\log(V)$. Overall the cost can be expected to grow like $(E + V)\log(V)$.

This is fairly reasonable in many cases, and ordinary heaps are easy enough to implement (and I will review them and their properties in a moment). However some graphs will be fairly sparse (so $E \ll V^2$) but not sparse to excess (eg $E \gg V$). In such cases a priority queue based method is still better than cruder linear searching, but the $E \log(V)$ term in the cost function dominates. I will be showing how to avoid this, to give a complete algorithm with costs proportional to $E + V \log(V)$. To give fair warning, the overheads involved will make this method unattractive in most if not all real applications, but that should not dampen enthusiasm for its aesthetic appeal or theoretical importance.

2.1 Ordinary Heaps

First, a review of (ordinary) heaps. A heap will be a binary tree with values stored in each node, such that the smallest value in the entire tree is in the top node, and each sub-tree also satisfies the heap property. I will generally want a heap to be a pretty well balanced binary tree, so that if it has n items in it its depth is around $\log(n)$[1]. The Part IB course explained how heaps could be represented by storing the values in a simple vector and using address arithmetic to give implicit pointers up and down the heap. Thus a value stored at offset k in the array might have child nodes at location $2k + 1$ and $2k + 2$. For the purposes of this course I will back off from such cunning, and think of heaps as made out of collections of separate nodes with explicit pointers linking them all. It will be necessary to have pointers both from parent to child nodes and back from a child to its parent. The issue of allocating space for the nodes is something I will ignore. The overhead of leaving space for all the explicit pointers and the pain of updating them all in step will also be something swept under the carpet by saying that any local update operation will have cost $O(1)$ and so is within a constant factor of the cost of any cleverer scheme.

With heaps as considered here it is useful to list the operations that can be performed, and remind ourselves of the associated costs. In each case n stands for the number of items in the tree:

[1] Of course all my logarithms here are base 2.

make-empty-heap	$O(1)$
identify-smallest	$O(1)$
insert-new-item	$O(\log(n))$
remove-smallest	$O(\log(n))$
decrease-key	$O(\log(n))$
delete-arbitrary	$O(\log(n))$
heapify	$O(n)$
union	$O(n)$
search-for	$O(n)$

I need to elaborate a little on the expensive items at the end of my list. The **union** operation is to take two heaps (of total size n) and make a new single heap out of their nodes. It can be done in the time indicated by flattening the two heaps into a single unordered list of length n and then heapifying it. The **search-for** operation starts with a heap and a key, and scans the heap to find an entry in it that matches the required key. Balanced binary search trees can do this in time $O(\log(n))$ but heaps are no better than linear lists in this respect. The **decrease-key** entry is for the operation needed in Dijkstra's algorithm where the priority of a node gets decreased and the heap condition is then restored.

2.2 Binomial Heaps

As a step towards my ultimate goal (Fibonacci heaps) I will describe Binomial Heaps which have the property that the cost of the **union** operation drops to $O(\log(n))$, but to partly compensate the cost of identifying the smallest item in the heap jumps to $O(\log(n))$. Fortunately this latter is usually not a problem, since after identifying the smallest item in a heap most applications remove it (with $\log(n)$ cost anyway).

I lead into Binomial Heaps through Binomial Trees. The Binomial Tree B_k will have 2^k nodes and depth exactly k. This sounds rather similar to the situation with regular binary trees! But a Binomial Tree is not binary—in particular the top node of B_k will have k children. These children will themselves be structured as Binomial Trees, and listed from left to right will be B_{k-1}, B_{k-2}, $\ldots B_0$. If course B_0 is just a node with no children at all. Although the fact does not worry me too much at present the number of nodes at depth i in such a tree is ${}^kC_i$, a Binomial Coefficient. Count the total nodes in B_k and it is $(B_{k-1} + \ldots B_0) + 1$ (the $+1$ on the end is for the parent node that holds it all together. It is then easy to see that this is compatible with the tree containing 2^k nodes. It is also useful to observe how one makes B_{k+1} out of B_k. The larger tree is just obtained by extending the smaller one by adding a second copy of it on the front of the list of sub-trees. This gives another way of setting that the size of B_{k+1} is just twice that of B_k.

Now for Binomial Heaps. Start by demanding that the values stored in a Binomial Tree satisfy the heap property, ie that the smallest value in any sub-tree is in the root of that tree. Thus the smallest value in the whole tree is right at the top. Now a Binomial Heap is a list of Binomial Trees, heap ordered and where no two of the trees are the same size. I think that perhaps an easier way to understand this is to take a complete Binomial Tree and ignore the top node (so that the children of that node now form a set of binomial sub-trees), and to allow some of those subtrees to be omitted. In some other sorts of trees one arranges that if the number of items to be stored is not a power of two then it is leaf nodes that are left out to get the numbers correct. Here we omit top-level branches.

I should observe that given any particular size n there is a single way of structuring a Binomial Heap of size n. This is because each of the trees it is made up out of has a size that is a power of 2, so we can just select which trees should be included by looking at the binary representation of the number n. Since I have ordered my lists so that larger sub-trees come first the "bits" in this binary representation will be stored with the highest power of 2 to the left.

Now a small observation for which cautious readers will want to write out a proper justification. A Binomial Heap containing n entries will have height about $\log(n)$, and this remains so even if the representation used uses linked lists to chain together the children of a node and traversals along such a chain are counted as contributing to "depth". This is part of why the largest sub-tree was stored first (to the left, and at the head of the list of children). This important fact will be what ensures that operations on the heaps is efficient. It should also be clear that the top-level list of binomial-sub-trees is of length bounded by $\log(n)$.

I can now consider the implementation of all the heap operations:

make-empty-heap: Easy! Clearly still $O(1)$.

identify-smallest: It is now necessary to scan the top-level list of trees. Each is individually heap-ordered so has its smallest item at its top. So to find the smallest item in the whole tree costs $O(\log(n))$, the length of the top-level list.

union: Most other operations are implemented using a union operation, so that is what I will describe next. Consider two Binomial Heaps as if they were binary numbers with the sub-trees within them standing for the individual bits. The algorithms required will then be just the usual one for addition, taking care with carry operations and arranging that the digits in the answer are built out of the trees in the inputs. It will be necessary to ensure that trees remain heap-ordered throughout the process. I happen to have specified my version of Binomial Heaps with the most significant bit stored first, which is a minor inconvenience here, but easy to program around. Following on

from hardware courses I observe that all I need to do is define a half-adder. I can make a full-adder out of two half-adders and go on from there. A half-adder takes in two "bits" and generates a sum and a carry. To make this convenient to code I will now insist that each node in a Binomial Tree is labelled to show its height. The height is then just an indication of the bit-number that it stands for. The only interesting case in implementing a half-adder is when both input bits are present, so they must be combined to produce a carry bit. If the bits are Binomial trees with structure B_i the carry that is generated needs to have structure B_{i+1}. However, as explained earlier, this can be achieved by just putting one tree at the head of the list of children for the other. This is clearly a $O(1)$ operation. A simple check on the top values in each tree shows which one should be on top and which below to preserve heap ordering within the new tree. Thus the code to implement a half-adder will run (easily) in $O(1)$ time. And since there are $\log(n)$ digits to process the entire union operation can be achieved in time $O(\log(n))$. I think this is clever!

insert-new-item: Create a one-node Binomial Heap (easy) and form the union between it and the original one.

remove-smallest: The smallest node in the entire heap is at the top of one sub-tree. First remove that subtree. Finding it and splicing it out of the top-level list costs $O(\log(n))$. Now look at the removed tree. Discard its root node (the one that was to be removed anyway) and what is left is a Binomial Heap (it happens to have all the possible sub-trees that it could). Form the union of this with the main heap, thereby putting back the items that were taken out along with the smallest element. Looks a bit messy, but the cost is still $O(\log(n))$.

decrease-key: This only involves adjusting values within one of the constituent binomial trees, and is just about the same as the operation needed to restore heap-ness when a value in an ordinary heap is reduced, so it still costs the same.

delete-arbitrary: Reduce the value in the arbitrary node to $-\infty$, thereby making it the smallest item in the whole heap, and then remove it.

The effect of all the above is to tame the cost of forming the union of two heaps (eg two priority queues). This is not what is wanted for the Dijkstra shortest-path algorithm, but can be very useful in other circumstances and the explanation helps prepare the way for the next topic.

2.3 Fibonacci Heaps

These are yet more complicated to implement than Binomial Heaps, but will end up with all the heap operations that do not involve deletion having $O(1)$ cost. To be specific, the costs are

make-empty-heap	$O(1)$
identify-smallest	$O(1)$
insert-new-item	$O(1)$
decrease-key	$O(1)$
union	$O(1)$
remove-smallest	$O(\log(n))$
delete-arbitrary	$O(\log(n))$

This seems stunningly good, and will be of great benefit to many algorithms where heap updates are much more common than heap deletion (as if the case with Dijkstra's algorithm on most reasonably sparse graphs, say ones where $E = V\log(V)$). There are two caveats that I have to make here. The first is that the times quoted for ordinary and Binomial Heaps were worst case ones, while the ones given here are **amortised** costs. This is still worst case analysis in some very real sense, and does not depend on the data being processed, but it bounds the average time per operation over a long series of operations. Thus it allows an implementation to queue up work so that most operations only involve the very cheap step of adding a request to a queue, but then occasionally one step will trigger major calculation that rearranges data structures to incorporate the batch of changes. An algorithm has amortised cost $O(1)$ if the average time per individual use of it over a worst case sequence of calls is bounded. Algorithms where the only good computing time bound is an amortised one may not be suitable for use in systems with a real-time response requirement, even though they may guarantee excellent long term average performance.

A Fibonacci Heap is a collection of trees. For Binomial Heaps all the nodes were linked by one-way lists. In the Fibonacci case two-way lists will be used for all links, which makes stepping forwards and backwards easy as well as allowing for the splicing in of extra items. So the top level of a Fibonacci Heap is a double-linked circular list of nodes. Each node contains a pointer to one of its children, and the other children form a double-linked circular list threaded through this primary child. A node will contain a field that records how many children it has, and a flag bit which is used to help the system know when and how it needs to re-structure its heaps to keep them from getting too straggly.

The heap as a whole has a pointer to the place in its top-level circular list where the subtree with the smallest top-node lives.

I am going to omit proofs that Fibonacci Heaps have the amortised costs indicated for them, and will rely on the fact (which I will not prove) that all the

operations on them that I perform leave them in a state where no node has more than $\log(n)$ children (where there are n nodes in the whole heap).

Suppose that the only operations performed on our heaps are

1. make-empty-heap
2. insert-new-item
3. identify-smallest
4. union
5. remove-smallest

then the Fibonacci Heap will be structured just like a corresponding Binomial Heap except that the order of the trees at the top level can be arbitrary (and indeed there can be several trees of a given size present, but only at the top level) and the order of the subtrees that make up parts of the constituent Binomial Heaps is also arbitrary. This is why you have learned about Binomial Heaps first. Clearly leaving the order of all the sub-trees unconstrained saves some time building the lists but makes other operations harder. The idea behind Fibonacci Heaps is to strike a good balance—trees are left in a mess for as long as is compatible with the final clean-up remaining sufficiently cheap.

Making an empty heap is easy and clearly has unit cost. To add a new item a new fragment is created and spliced into the top-level circular list. If the new item has a smaller key than the previous smallest item then the relevant pointer is changed. After a long sequence of inserts the heap would end up as a long thin chain. Nodes inserted in this way have their flag bit set to **false**. Inserting a new item in this way also clearly has unit cost. The minimum item in the tree is always directly identified, so accessing this also has unit cost.

To form the union of two Fibonacci heaps the two top-level chains are just linked together, and the minimum pointer of the combined heap becomes one of the minima in the two original heaps. This again tends to lead to long straggly lists, but as an individual operation it remains cheap.

That leave the remove-smallest operation, which is where all the mess arises. It is necessary to consolidate the heap as a whole to improve the extent to which it is balanced. After a long sequence of insert operations the first call to remove-smallest may appear fairly expensive but it will then clean things up so that subsequent calls are cheaper.

The first thing that is done is that each child of the smallest node is moved up to be a tree in the top-level cycle. Then the smallest node itself can be removed easily enough. The cost of this involves a small operation on each child of the smallest item in the heap. I asserted earlier that the largest fan-out that could arise

would be $\log(n)$ so this is still reasonable. The structure is now still a Fibonacci Heap, but at this stage it the top-level subtrees will be consolidated so that at the end of the deletion operation no two subtrees will be the same size.

To ensure that all subtrees end up the same size we use a working array of length $\log(n)$, and start it off with all elements NULL. Then each item in the original heap is considered in turn. Each tree is marked with its height (if I had not mentioned that before I do so now, it is clearly not a difficult thing to arrange), and all heights are less than $\log(n)$. So we try to post each sub-tree into one slot in the working vector. If when we find that the vector element concerned is already full we have just found two trees whose sizes match. In that case the two trees can be combined (almost as for Binomial Trees) into a single one of double the size (and hence height just one greater) and this can be inserted into the next array element up. Almost the only thing that one has to take care of is to ensure that whenever two subtrees are merged the one with the smaller root ends up on top. At the end of a scan of all the sub-trees we end up with the working array holding a collection of trees. The algorithm terminates by collecting these and forming a circular two-way list out of them, with the one of them that has the smallest key specially identified, and this is then the reconstructed Fibonacci Heap.

If this is done just after a large number of items have been added to the tree one at a time the cost will be n steps to consider each individual node, plus $n/2$ to handle the first stage of "carry" operation, plus $n/4$ times for a second-stage carry and so on—all in all the number of individual steps is proportional to just n. But in the spirit of amortised analysis we can view this apparently linear cost as being shared between the n operations used to build the heap up to now, and thus as contributing $O(1)$ to each such heap-building step. After the heap has been consolidated it will consist of trees all of different sizes, so there will be at most $\log(n)$ such trees and subsequent remove-smallest operations will each have costs limited to this amount.

Note that it was important to have a bound on the largest size tree we would have in the whole heap so that the working array could be set up, and necessary (for efficiency) that the array end up having size logarithmic in the total number of items stored.

To delete an arbitrary node from a Fibonacci Heap we can first decrease its value to $-\infty$ and then perform the remove-smallest operation, so all I need to worry about now is the process of restoring the structure of a Fibonacci Heap after a value stored therein has been decreased. However this is messy! After it has been done the Fibonacci Heap is no longer quite a set of (unordered) Binomial Trees. First observe that if the node being altered is the root of its tree, or if the new value is still larger than the value stored in its parent node, nothing very complicated needs doing. The only problem is when the new key is smaller than that in a node higher up than the node being altered. In this case we lop the

changed node (with all its children) out of the place in the tree where it originally sat and put it in at the top level as a new tree. As such this is clearly very easy to do. The potential problem with it is that it deletes a child from an arbitrary place within some other tree, and can thus leave that tree with less fan-out than was desirable, in particular eventually leaving that tree much too straggly. The depth of the tree could reduced overall by taking all the nodes on a chain upwards from the changed node and moving them all up to the top level. But doing this all at once would mean that the cost of changing a key was much greater than $O(1)$. As a compromise it is again useful to observe that $1 + 1/2 + 1/4 + 1/8 + \ldots = 2$ and so arrange that successive bits of this rearrangement are done at frequencies that roughly reflect the above geometric progression. This is where the flag bit mentioned before comes in. We arrange to set the flag bit whenever a node loses a child (through this restructuring step). The flag is re-set whenever node is moved up to top-level. With this flag in place we can detect the **second** occasion when a node loses a child, and let that cause the node to be relocated to the top level in the heap. The effect is rather as if the chain of flag bits leading up from a leaf in a tree towards the root can act as a binary counter—on average the number of carry operations (ie reorganisations) will be small even if occasionally a lot happen all at once. The overall result will be that each decrease-key will perform $O(1)$ steps of moving a node to the top level, and so ends p with cost $O(1)$.

Note that a sequence of decrease-key operations can lead to bits of tree that are in fact just linear lists. So now that the sub-trees involved are not arranged as if they were Binomial Heaps I ought to go back and re-think my heap consolidation algorithm. What is in fact done is to use the degree (ie number of children) of a node as the basis for the merge operations to be done. If a node has lost some children it will be treated as if it were a smaller tree, but nothing very untoward happens as a result. The main clever bit of analysis is to show that even after a tree has been subject to an arbitrary collection of decrease-key and consolidation steps the greatest number of children that any node can end up with is proportional to $\log(n)$. A proof is worked through in Cormen et al[1], as are proper justifications that all the operations involved have the (amortised) computing times claimed here. But in each case I think the full details are too long and messy to be included in this course or its notes.

Fibonacci Heaps are so called because a bound on the sizes of bits of the trees turns out to be a Fibonacci number.

2.4 Consequences

The ability to do decrease-key in $O(1)$ time can be a great help in many algorithms of potential practical use. As well as shortest paths it has direct application to minimum spanning trees, bipartite matching and other problems. The only thing that

can really be said against Fibonacci Heaps is that the overhead of keeping all the doubly-linked circular lists and up and down pointers, and flag bits and degree counts up to date can look quite painful, and so in many real cases accepting the extra $\log(n)$ factor that a simple heap would suffer is in fact sensible. The challenge that that leaves you with is to design a new and better data structure having the compactness and absolute speed of ordinary heaps and the superb asymptotic performance of Fibonacci ones.

3 An Introduction to Kolmogorov Complexity

Overall Kolmogorov Complexity is a subject area that has only started to become widely known quite recently, and which is full of seriously delicate traps and pitfalls. It allows one to discuss the "complexity" of finite objects, or to consider whether individual events are "random". Using it it would be possible to make a precise form of the statement that you would be amazed if the National Lottery turned up the set of numbers $\{1,2,3,4,5,6\}$ rather than some more scrambled looking bunch. What is better, it would potentially allow you to design a rather complicated bet you could make which would result in you winning if the lottery numbers were fixed (in any way!), and lose if they were fair and random—an admirable bet for those of a paranoid disposition. I am going to produce an introduction to the techniques of the subject, and illustrate it by showing how it allows a Computer Scientist, using obviously Computer Science lines of argument, to prove something that you would normally think you needed a number theorist to sort out. To be specific I will prove a lower bound on the density of prime numbers, showing that primes are in fact quite common. The result I will end up with will not be as precise as the ones that the Mathematicians have (in fact by quite a long way), but the way in which I conduct the proof is not going to have to call upon any difficult number theory techniques or results, instead it will rely on an understanding of Turing Machines and data compression algorithms! I selected this example for inclusion in the course because it gives a neat illustration of the fact that Computer Science analysis techniques are different from but sometimes as elegant and powerful as Pure Mathematics ones—and if I can make a successful application of Kolmogorov Complexity to this problem I might come to believe it will have many other uses.

3.1 Perfect Data Compression

To start off with I should consider data compression. I will consider all data as consisting of strings of bits, so the size of some data is just the number of bits used. Well actually it turns out that even that simple-sounding statement is

probably dodgy, and in places I will need to think again! When you take out your pen and write down a string of bits you will concentrate on the fact that each bit is either a '0' or a '1'. But when you stop writing you will leave a gap on the paper that shows where the bit-string ends. This gap is itself not a bit and previously when looking at measurements of data we probably only worried about the genuine direct data. But implicitly there will be a "data length", and the information conveyed by that must not be ignored. An extreme case in point would be if the valid binary strings used were to be restricted to "1", "11", "111", "1111", and so on, counting in unary. Then all the real information content is contained in the (implicitly recorded) length. Some styles of data will manage to encode length or termination information as part of the honest data, and this will generally be considered good. Two common examples spring to mind: individual characters coded using a Huffman Code are self-terminating, and text files under MSDOS may be represented as a string of bytes, with the end of file marked by the special "character" control-Z[2]. In the C language strings are terminated by an explicit zero character and so are the bit-pattern representation of them (including the final zero) is self-terminating.

Now for data compression. I will define a data compression scheme as a pair of mappings between bit-strings. I will generally be interested in compression that starts with bit-strings where the length is implicit. Thus the **compression** procedure must be able to accept any of the 2^n distinct bit-strings of length n and it turns it onto some other string, which ideally would have length less than n, at least a lot of the time. The **decompression** process will only ever be fed bit-strings that have been generated by the compression process and is entitled to misbehave in an arbitrary fashion if given a string that was not created in that way. But if given the compressed version of some data it must always reconstruct exactly the original data string, including knowledge of its length. Despite the fact that the mapping process is called *compression* it is probable that some input strings will grow when "compressed". Observe

1. No compression algorithm can cause every input string to get mapped onto a strictly shorter compressed form;

2. If a compression scheme has the property that all input strings are either shortened or stay the same length (so at least none get strictly worse) then none actually end up shorter;

3. You can't even have a compression scheme that manages to reduce the size of half of all possible input strings.

[2]this explanation about the file structure is of course not totally true, but it will suffice for now

These results follow by counting how many bit-patterns of length n are available and observing that there have to be enough distinct compressed patterns to allow distinction between all possible inputs.

As an aside I might indicate what this tells us about compression utilities such as "zip". The very high compression seen when such utilities are used on typical files indicates that the patterns of data in such files is very special in some way. The fact that zip never seems to cause any file to grow is only nearly true—it can spot files where its data compression techniques would not make good headway and store the raw uncompressed data with a short prefix that can tell the decompression process just to copy what follows. Thus it can arrange that in its worst case the compressed file is only a few bytes longer than the original, while in cases that are very sparse in the space of all possible input bit-patterns but which happen to be remarkably common in human-generated files it can achieve compression ratios of (often) 3:1. This is not in conflict with my more theoretical argument.

Refinements of the counting argument allow one to show that only a really tiny proportion of input strings can be compressed to a significant extent. In particular at best half could save one bit, or a quarter could save two bits, or an eight could save three bits. Strings that can be compressed to half their original length are stunningly rare.

Now I should consider the concept of an optimal compression procedure—one that will take any string and produce the very best possible compressed version subject to the constraint that there must be a single associated decompression algorithm. At first sight it seems very improbable that a universally optimal compression procedure can exist, since it seems probable that compression techniques should depend on the class of strings to be handled. There is a small bit of sleight of hand that gets around this! Imagine we have n different compression methods, called $C_1 \ldots C_n$ and each works best in different circumstances. We can make a composite method by trying each of C_1 to C_n and compressing with the one that works best. The resulting output string then has a prefix of $\log(n)$ bits stuck on the front as a signature of which scheme was used. Across any very large set of strings the constant number of bits that this introduces is unimportant and can be ignored. So with this in mind we agree that one compression process can only be considered better than another if not only does it compress some strings to strictly shorter lengths, but if there is no constant bound to the amount by which it does better than the method it is being compared against.

With this way of comparing compression methods it turns out that there is a universally best method (or to be more precise, a whole family of mutually equivalent "best" methods). This method is (also amazingly) very easy to describe. A string S gets compressed to the shortest program that will print the bits of S and then stop.

The Computation Theory Course shows, in effect, that any worthwhile programming language can be used to simulate any other, and only a finite sized program is needed for the simulator. To be specific it suggests that Turing machines might form a sensible concrete way of expressing the programs we need here, and it demonstrates that the description of any particular Turing Machine can be reduced to a numeric code and hence a string of bits. Thus when I feel pedantic I will say that the best compressed form of any string is the shortest string of bits that represents a number, which in turn describes a Turing Machine, such that that Turing Machine would, if started with an empty tape, ends up writing the original bit pattern to the tape and halting. Please note that until the machine halts it will not be possible to be certain that it has finished generating output.

The universal nature of Turing Machines then shows that any computable decompression strategy can be implemented this way with only a fixed-length burden. This burden is the encoding (as a Turing Machine description) of the decompression strategy, which is then simulated. So this scheme really does deliver as good compression (to within a constant additive amount) of any compression scheme possible. There is a slight difficulty, in that identifying the shortest Turing Machine is a hard problem. Well actually it is undecidable. But it is certain that a shortest machine does always exist, and I will therefore ignore the slight embarrassment of having the halting problem intervene between me and a neat implementation of all this!

3.2 Complexity Measures via Compression

I now define the Kolmogorov Complexity of a string as the length of the ideally compressed version of it. This value will be well defined to within an additive constant. Observe with some surprise that the complexity if the infinite string that starts

$$2718281828459045235360287471352662497757247093699959 5\ldots$$

is finite (in case you do not recognise it this is supposed to be the digits of e, and a very definitely finite program can be persuaded to print the digits one after another without end. The same would be true for the digits of $\sqrt{(2)}$ or π). However very nearly all strings will have Kolmogorov complexity just equal to their length[3]. This last fact turns out to be the key that makes this complexity measure useful. All sorts of jolly results can be demonstrated by relying either on the fact that in-compressible strings exist, or the stronger assertion that almost all strings are incompressible. I will demonstrate this on the "density of primes" problem.

[3]Length, that is, when viewed as a bit string.

3.3 The density of Prime Numbers

The proof I will produce here starts by imagining that primes are rather rare, and works by demonstrating that if that were so it would give us an opportunity to represent a number n in fewer than $\log(n)$ bits. Kolmogorov complexity tells us that good compression is very uncommon, and we deduce that the density of primes must be sufficient to make the compression process that I propose worthless. I should flag this section of the notes with a slight warning, to the effect that I will **not** be filling in all the formal details. The very heavy and expensive Handbook of Computer Science[5] or an even more specialised book specifically on Kolmogorov Complexity (cited in the Handbook, but I am not going to reference it directly here!) will fill things in with full precision for people who need that.

Suppose now that the nth prime is $p(n)$. I am going to think of $p(n)$ as an increasing function of n and will want to place a bound on how rapidly it can increase. Next consider an arbitrary incompressible integer k. Here calling it "incompressible" means that the ordinary bit-string that denotes it using binary notation (of length $\log_2(k)$) is about the shortest possible way of describing it. Remember that almost all bit-strings (and hence integers) are incompressible so I have not limited myself in a severe way. I now consider an alternative way of describing the number k. Split it into its prime factors, and identify the largest of these. Call that $p(i)$. Call the co-factor q, so that $q = k/p(i)$. Now try to represent k as the ordered pair (i, q). The reason this is worth trying is that if primes are very rare then $p(i)$ will be a much larger number than i, and so using i to stand for it will save us a lot of bits.

Observe that (if I ignore the issues of rounding to integer values) that $\log(k) = \log(p(i)) + \log(q)$. That is nothing more than the usual rule about how to use logarithms to multiply things together. At first sight it may seem that I can represent the pair (i, q) in $\log(i) + \log(q)$ bits by just writing down the bits for i then the ones for q. However this is where the discussion about encoding the lengths of strings cuts in. If I just write down the bits for i and then the bits for q I have no way to tell where one ends and the next starts. To cope with that I will start off my compressed string with a number representing $\log(i)$, the number of bits I use to represent i, so that I can tell. Thus in some sense I will be using a representation based on $(\log(i), i, q)$, and the separation point between i and q is now well specified. But oh dear, the boundary between where I specify $\log(i)$ and where I start i itself is now murky. Let's play the same game yet again, and try writing the script as $(\log\log(i), \log(i), i, q)$. And at this stage cop out and observe that the first number, which is now liable to be pretty small, can be written in binary but by using two characters per bit we can include an explicit termination mark in it. To illustrate this last point I will show how the number 5 can be represented in a self-terminating way. In natural binary one would have 101, but to make things

self terminating I will interleave extra digits 1**0**0**1**1 where the marker **1** first appears to indicate that the end of the number has been reached. I will now try this out as a potentially compressed representation of my original number. I end up with $2\log\log\log(i) + \log\log(i) + \log(i) + \log(q)$ bits. And because even though this seemed like an interesting compression process Kolmogorov complexity tells us it can not work, we know that this bit-length is at least as long as the length of our original number, ie $\log(k)$, which we saw earlier was just $\log(p(i) + \log(q)$. To cope with the additive constant by way of slack in the complexity analysis, I will stick in a stray constant V, and end up with an inequality:

$$2\log\log\log(i) + \log\log(i) + \log(i) + \log(q) + V > \log(p(i)) + \log(q)$$

and this very easily rearranges to give

$$\log(p(i)) < \log(i) + \log\log(i) + 2\log\log\log(i) + V$$

Unwinding the logarithms, and letting $\log(A) = V$, this gives

$$p(i) < Ai\log(i)\log\log(i)^2$$

At present this result only applied when $p(i)$ is the largest prime factor of an incompressible number. But since there are an infinite number of primes there are lots of numbers available whose largest factor is itself quite large, an in particular (though I will not fight to prove it formally here) enough that one can be certain that some of them are incompressible.

It is more common to consider the number of primes less than some integer k. This count is usually called $\pi(k)$ and what has been shown here is equivalent to showing that $\pi(k)$ is close to being bounded by something proportional to $k/\log(k)$. You should note that mathematicians manage to do better, and can show that for large k, $\pi(i)$ is rather close to $k/\log(k)$. In this case the logarithm has to be a "natural" one to base e. This is better than my result firstly because it turns the inequality into (which provides just an upper bound) into something that gives both upper and lower bounds. They also get rid of the $\log\log(i)^2$ term, and show that the constant A can be taken to have the value 1. A suitably gentle and maybe old fashioned book that does all this the mathematicians' way is Hardy & Wright[3], which is certainly not required reading for this Part II course.

I use this example to introduce Kolmogorov complexity because I think the proof is short and elegant, and it is pleasing to see computer science nibbling away at the corners of mathematicians' territory. However the idea of proving a lower bound on something by showing that beating the bound would allow you to compress some strings rather well has many applications, and it is one of the best and most general approaches to proving lower bounds on the costs of solving

problems. The handbook[5] applies this to show that a (one tape) Turing machine must take at least kn^2 steps to accept a palindrome of length n [k is some unknown constant], that adding n integers together must take time at least proportional to $\log(n)$ on a certain (quite general) class of parallel computers, and that the resources needed (time and space) for multiplying boolean matrices together must in general grow at least as fast as kn^3 for n by n matrices.

3.4 Consequences again

At first sight the idea of optimal compression is an entertaining one but slightly silly, because finding optimally compressed versions of particular strings is undecidable. However that does not in any way prevent it from giving a powerful and unexpectedly practical way of proving computing bounds. It is important to understand that even though identifying the best compressed form of a string is undecidable, the best compressed form is well defined and unambiguous.

More elaborate versions of the same idea consider compression and decompression subject to resource bounds, or the compression of one string supposing that another one is already known to be available.

The use in connection with finite "random" sequences starts by defining a string to be random if it is not compressible. In this sense one would view an outcome from the National Lottery that was "31, 4, 15, 9, 26, 5" as dubious because the pattern of digits can be described too neatly by reference to the constant π. A string quite that short might (just) be accepted as an accident, but if the next week followed on with results based on the next few digits of π I think one would be entitled to be seriously concerned. By putting Kolmogorov complexity in the context of probability it is possible to design bets, that are in effect of the form "I bet that the string of numbers you give me can be compressed really well" where the average outcome is well balanced if the numbers selected are indeed truly random, but which would deliver big returns if the person selecting the numbers cheated in **any** way that amounted to having some mechanisable, computable function causing diversion from true randomness.

4 Probabilistic Algorithms

Having used a number theory problem to illustrate Kolmogorov Complexity, I will use another to motivate a consideration of probabilistic algorithms. To be specific I will look at the problems of identifying prime numbers and of finding the actual factors of numbers that are known to be composite. These tasks have had a much increased visibility to computer scientists since the introduction of certain Public Key encryption methods which rely for their security on the fact (well at present

it seems to be a fact!) that it is possible to find large primes and multiply them together fairly fast, but finding the factors of a large number is usually very hard. Right at the start of this discussion I should make it clear that such encryption and authentication applications will be working with numbers that have say 200 (decimal) digits. Ordinary 32-bit "machine" integers have such a limited range that brute force can handle them with hardly any need for fancy algorithms! Most of what is discussed here can be found in Cormen et al[1]. An alternative presentation covering much the same ground is in Knuth[4] volume 2. At least last year there was a Mathematics Part III course on Computational Number Theory, given by R G E Pinch, and when I last looked his lecture notes were available via his home page, which itself was reachable via the Pure Mathematics Department's web page. Those seriously interested in proofs or in an understanding of just where the current practical limits to our ability to factorise are could read more there. Obviously the Computer Science Part II course on Security will elaborate on why the issues I discuss here are important ones.

4.1 Naive Algorithms and their costs

Given a number N it is very easy to look for factors by trying trial division by 2, 3, 4, 5, ... until either a factor is found or until it becomes clear that N is prime. A more idealised version of this method would just try dividing by primes, but to achieve that a list of primes would be needed and for large N that becomes infeasible. It can however be both sensible and convenient to avoid an attempt to divide by a number that is even or a multiple of 3 or 5. If N is composite then it will have a factor that is no larger than $\sqrt{N}$, so at worst the number of trial divisions needed grows as $\sqrt{N}$. But the size that we use to measure N will be the number of digits used to display it, and measured in terms of this trial division is a procedure with exponential costs (however fast the division process itself is). To get a feeling for this observe that trial division can find factors of any 32-bit number quite rapidly (less than 64K trials are needed), but for 20 (decimal) digit numbers one would be close to the limiting capabilities of the fastest computers. This method is both a test for primality and a way of finding factors. It will be at its slowest when investigating a prime. The next section reveals that (at least probabilistically) identifying primes can be done **much** faster, and there are distinctly better factorisation methods available too.

4.2 The Miller-Rabin Test for primality

A treatment of primality checking that was going to contain proper proofs of the validity of algorithms would normally work through two or three checking algorithms before getting around to the Miller-Rabin test. These earlier methods have

the defect that certain numbers can fool them to the extent that they can not detect that those particular numbers are composite however hard they try (the tern "Carmichael Numbers" creeps in here). The process at the heart of the Miller-Rabin is better behaved. For **any** prime input it will always report back that the input is indeed probably prime, while for **any** non-prime input it will have at least a 3 in 4 chance of being able to detect that fact, and hence at worst a 25% chance of indicating that the number may be prime. A critical issue for computer science in this is the interpretation of those probabilities. In this case the test works on not just the input number N but on a "random" value a. The 25% bound on the probability of mis-reporting a composite number as "possibly" prime is not dependent on the value of N and only relates to the range of possible a values that can be selected. Thus if a true random process is used to select a value for a it will be a genuine probability on the outcome.

The process is amazingly simple. Start with an odd number N, and select a random a in the range from 1 to $n-1$, with all those possible values of a equally likely to be picked. What it does is to form the value $a^{N-1} \bmod N$. This power can be computed by a process that spends most of its effort repeatedly squaring values starting with a. Because N is odd the exponent $N-1$ will be even and the last few steps of computing a^{N-1} will all be just squarings. The test reports that N is composite if either

1. $a^{N-1} \bmod N \neq 1$

2. **or** somewhere along the sequence of intermediate results there is a step where two consecutive values are b and b^2 such that $b \neq 1$ and $b \neq -1$ but $b^2 = 1$ where all arithmetic is being performed modulo N.

The code to raise a number to a power using repeated squaring[4] is easy to code, and adding the tests both at the end an on intermediate values is pretty trivial.

The final part of the complete Miller-Rabin algorithm is to apply the above "strong" test several times for different independent random values of a. Then in N is composite each test has an independent 3 in 4 chance[5] of spotting that fact. After (say) 30 trials the probability that something that still seems to be prime is in fact composite will then be bounded by 2^{-60}. Many people will believe that the probability of an undetected error in the program, or undetected hardware failure in the computer running it, or undetected transcription errors in the result, or some

[4]...arranging that if $N-1 = 2^k v$ for some odd number v the calculation first works out a^v however it wants and then squares the result k times.

[5]Several write-ups of the method indicate that it has at least a 50% chance here. The reason they say that is that that is a remark that is tolerably easy to prove. The 75% bound quoted here needs more mathematics to justify it, but is both true and (for some numbers N at least) a sharp bound

other general foul up will be much greater than this, so the fact that the result is not **quite** guaranteed is something of a frivolity. Those who feel more neurotic could run 60 trials and thereby reduce the chance of error to 2^{-120}, or whatever. But observe that the mathematical theory that justifies combining the multiple tests depends on the random choices for values of a being properly random and all independent, and arranging that on a computer will be hard. A lengthy section in Knuth[4] discusses the issue of real and pseudo-random numbers. And the way in which plausible computer processes for generating random-looking sequences can end up failing to do that. Time has moved on and the particular concrete recipes for competent random number generation that Knuth suggests are now no longer state-of-the-art, but his general warnings and overview discussions remain valid.

Fairly simple computing-cost analysis of the Miller-Rabin algorithm shows that if we are prepared to be content with a probabilistic test for primality it gives us one that uses an amount of time $O(s \log^3(N))$ where s is the number of random trials to be used. This is pretty good, and when combined with what we know about the density of prime numbers means that it is acceptably cheap to find really quite huge primes.

Note carefully that the Miller-Rabin test can certify in an absolute way that a number is composite (though it then does not actually exhibit the factors), but it is constitutionally incapable of giving an absolute proof that a number is prime. Such absolute certainty calls for quite different algorithms, which often call for the complete factorisation of $N - 1$. See Knuth or Number Theory books for more information!

4.3 Pollard Rho

If we have discovered that a large number is composite then finding its factors can still be very hard. But an unsettling issue is that we can not easily **rely** on it being hard, because at least in some cases a factorisation will just drop out trivially[6]. As a prelude to any high technology factorisation process it is prudent and normal to use trial division to detect any factors that are less than some predefined limit, typically a few million.

A simple-to-code and tolerably competent procedure that will often find factors using around $N^{1/4}$ arithmetic steps, while ordinary trial division needed up to $N^{1/2}$ again uses random numbers. This time there is not going to be any guarantee of either success or of a particular running time bound, but the method works well enough in practise to be worth reporting. The code sketched below should be imagined to be running on a computer where integers can be several hundred dig-

[6]Consider the special case where the input value happens to be a power of 2!

its long. It starts with a random initial value form x and forms a pseudo-random sequence using the crude recurrence $x_{n+1} = x_n^2 - 1$. Every so often we record a value of x in y, where "every so often" follows the sequence 2, 4, 8, 16, ... as indicated by the calculations involving k. The procedure exits if a gcd calculation manages to exhibit a non-trivial factor of N.

```
int pollard_rho(int N)
{
    int i = 1, k = 2;
    int x = random_number(0, N-1);
    int y = x, d;
    for (;;)
    {   x = (x*x - 1) % N;
        d = gcd(y-x, N);
        if (d!=1 && d!=N) return d;
        if (++i == k) y = x, k = 2*k;
    }
}
```

The procedure works (when it does, which is quite often) because if the sequence of values generated for x behaves randomly it is expected that an x-value will be revisited after around $\sqrt{N}$ steps. This just comes from looking at simple probabilities—if $\sqrt{N}$ independent random values are chosen from N possibilities it starts to be probable that two of them will be the same. But here once a value of x repeats the subsequent behaviour will be cyclic. If N is composite, say $N = PQ$ then the cycle will often have one length modulo P and another modulo Q, and this allows the gcd calculation to find one of P or Q. As coded above I return as soon as I spot one factor of N. Some people would continue running the loop in the hope of uncovering more factors from the same run.

There are collections of other approaches to integer factorisation, and for many of them the key parts of the algorithms are almost as simple to express as the rho method, but the theory and proofs that justify them are difficult. Probably the most interesting ones involve running many tolerably modest test calculations each of which can potentially deliver a small chunk of evidence about the factor structure of N, in such a way that when enough evidence has been collected a single large but manageable calculation can combine the partial results and reveal a complete factorisation. Using such an idea the biggest factorisations achieved to date have used several hundred MIPS-years of CPU cycles in all (using many computers linked by a crude e-mail communications web) and have cracked numbers with up to around 150-160 digits. Because costs grow exponentially with the number of digits in the input number N this means that factorising 200-digit numbers is still not on.

4.4 Theories about Probabilistic Methods

With the above prime number related algorithms any guarantees about performance or outcome depend on having a reliable source of true random numbers. If **any** algorithmically defined sequence of numbers was used instead that would be at least an outside chance that some especially carefully chosen number would relate so specifically to that algorithm that it would then systematically defy factorisation. This at least suggests that there might be a general value in studying the properties of Turing Machines that have been extended to provide some sort of source of true randomness.

The way that research in this area has progressed is through the definition and study of a number of complexity classes, similar in their various ways to the classes P and NP[7]. With randomness it turns out that there are quite a few plausible ways of defining associated complexity classes. I will describe a few here but neither prove relationships between them nor give definitive examples that illuminate the capabilities of each.

4.4.1 The class R

Without serious pain we can take any non-deterministic Turing machine that is about to solve a problem for us and adjust it so that all its calculations are finite and then so that all its calculations have the same length (we can pad any short paths with extra waste computation to achieve the balance). We can also arrange that every place where the NDTM has to make a choice that choice is purely binary, so each state will have either one or two successors. Beyond that it is possible to arrange that each possible calculation that the machine can make will have the same number of choice points on it. Why do all this? Mainly so that the possible behaviours of the machine are in a regular enough pattern that we can talk about randomness and probability with less pain. Note that since the costs associated with an ordinary NDTM are taken to be worst case ones, padding all computations up to this does not alter the overall pattern of what they can do.

With this understanding, the class R is the set of all problems that can be solved with an NDTM such that either there are no accepting computations or at least half of all computations are accepting. Because the style of NDTM being used is balanced it is possible to take each non-deterministic choice at random and the 50% ration of accepting computations turns into a 50% chance of eventual

[7] P relates to a fully deterministic Turing Machine and NP to one that has access to a truly inspired Oracle, and the classes include those problems that can always be solved efficiently by machines of the types considered. Of course a precise statement of what is meant by "solved" and "efficient" would require more than a footnote here, but was provided in last year's course on Complexity

success. Observe that the errors that this sort of machine can make are one-sided, in that the only thing it can do wrong is to fail to exhibit a solution to a calculation that in fact has one. It can never report a solution to a problem that does not have one! That one sidedness means it is sometimes useful to use the class co-R, where it will be the "yes" answers that can be wrong with probability up to 1/2 and the "no" answers that are totally reliable. The class R is the one that arose with the Miller-Rabin test for "is this number composite". Note however that this example neither serves to show that R is strictly larger than P, nor that the compositeness-test problem really needs randomness[8].

4.4.2 The class ZPP

This is the intersection of R and co-R. An alternative characterisation is that it corresponds to randomised calculations which always terminate, but which can they yield one of three outcomes, "yes", "no" and "don't know", where the probability of the last arising is less than 1/2.

4.4.3 Probabilistic Turing Machines

Consider machines where all calculations are the same length and contain the same number of choice points, and where all computations end reporting "yes" or "no". Now this is interpreted as a probabilistic TM if we say that the machine as a whole answers "yes" if more than half of the paths through it end on a "yes, and correspondingly. PP is the class of problems solvable in polynomial time using one of these. Note that PP is much less nice than R since it does not tell us what the probability of error will be, and the "more than half paths" that give the correct answer may really be "only very slightly more than half". Thus in general re-running a calculation can not be guaranteed to increase the probability of success very fast. To get around this we can define BPP as the set of problems such that we can have a chance of at least 2/3 (not 1/2) of getting the correct answer from a single run.

4.4.4 Game playing and Stochastic Turing Machines

Ordinary NDTM calculations are uncomfortably asymmetric. A computation succeeds if it can find any path through the machine. But it is also natural to want to consider cases where **all** paths through the machine lead to success (this observation is at the root of why it is usually the case that proving a solution exists

[8] Enthusiasts (?) may like to know that if the Extended Riemann Hypothesis holds then it is in P. But we computer scientists do not even know what the Extended Riemann Hypothesis is!

is in NP while proving one does not isn't). Game-playing strategies for two-player games provide a useful model for something a little more symmetric but still well constrained. To win, player A needs to find some single winning next move, while before conceding defeat player B needs to become convinced that all possible responses to that killer move hopeless. In Artificial Intelligence lectures you probably find out about the α–β heuristic for coping with just this. Alternating Turing Machines are ones where the meaning of "success" in a calculation is expressed in terms of the knowledge needed by players in such a two-person game. Stochastic Turing Machines combine this idea with random responses, and so consider the problem of winning a game against an opponent whose moves are made utterly at random. As with probabilistic machines it becomes useful to consider the class of problems (now games!) where from any initial configuration the first (non-random) player has a chance of either more than 2/3 or less than 1/3 of winning. The class of problems that can be solved this way is called AM, where the initials are for Arthur-Merlin, and it turns out to be related to the analysis of interactive proof verification!

4.5 Consequences yet again

Just talking through the complexity classes I have mentioned here could take a whole (long) lecture course. There are two major reasons for looking at them. Firstly they can often allow one to derive unexpected links between different algorithms for solving real practical problems, and help with cost analysis of those methods. The other important issue is that developing a rich set of different complexity classes (P, NP, NP-complete, co-NP, R, co-R, ZPP, PP, BPP, AM and in fact I have only scratched the surface here) can help focus our attention on the hierarchy that these problems give us, and on problems that are on the borderlines between them. It shows that even if we cheerfully accept that we think that P is not the same as NP that there can remain a large number of challenging and practical issues of the same style that still deserve serious attention.

5 Real-time Garbage Collection

This topic is included in an attempt to make the course as a whole well balanced. The various clever Heap data structures are traditional (if slightly elaborate) data structures, Kolmogorov complexity illustrates a modern analysis technique with a strongly computational flavour, while the discussion of Probabilistic methods looks back to the complexity course and sideways to discussions of security, artificial intelligence and beyond. I round things off by looking at the issue of algorithms and parallel hardware. It is clear that moving to parallel hardware re-writes

all the cost expectations for processes, so is cause for a fairly total re-think of almost all data structures and algorithms! Very great care has to be taken in this, however, since typically the costs of data distribution and synchronisation can exceed the operations that one normally thinks of as "computation". As may have been made clear in other courses on operating systems and concurrency, correctness in a parallel world is also a significantly nastier problem than in a sequential environment. I am just going to include one example here. It is included partly because its history illustrates the delicacy of algorithm design for true concurrency. A full write-up can be found in Dijkstra et al[2], and so I will not include a re-written summary version here.

6 Conclusion

As these notes[9] have grown it has become clear that there is a huge amount that I could cheerfully include in this course. The main message I want to get across is the enormous depth and complication there is within computer science, and that the study of algorithms did not become a finished-off dead subject just because (almost) everything about sorting techniques is now known. On the contrary, this area which must surely count as the very heart of Computer Science, continues to throw up challenges and problems both related to the very practical issue of finding the fastest way of solving certain real problems and to the theoretical issues of understanding the inherent relationships between problems, styles of computers and costs.

References

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

[2] Dijkstra, Lamport, Martin, Sholten, and Steffens. On-the-fly garbage collection: an exercise in cooperation. *CACM*, 21(11):966–975, 1978.

[3] Hardy and Wright. *An Introduction to the Theory of Numbers*. OUP, 1938.

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

[9] Please, whether you liked this course or not, find the Computer Laboratory web page and indirect through it to the course feedback area and record your thoughts. I already know that the notes were not available at the very start of the course, and for next year (when it will not be the first time the course has been given) that will not be a repeat issue, so maybe you can find other things to complain about!

[5] van Leeuwen (editor). *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*. Elsevier/MIT Press, 1990.