The next five lectures are devoted to the study of algorithms, that is methods to solve specific problems. Other lectures in CS1Bh, further down the course, will also discuss some specific algorithmic problems.

Algorithms is one of the longest standing areas of study in computer science and is continuing to develop vigorously. In the study of algorithms we consider step-by-step solution methods for problems rather than computer programs written in a specific programming language. This provides the engineer with a useful toolkit of methods that can easily be coded in the programming language they happen to be using for the task in hand. Generally, we deal with precisely defined methods but by dropping the need to consider a particular encoding in a programming language we can study very concise descriptions of the methods. The study of algorithms is strong thread in the CS curriculum at Edinburgh and it will continue throughout all years of your undergraduate course.

In studying algorithms we are often concerned to ensure their use of resources is parsimonious. The most common resources we consider are the time and space needed to solve a problem. In particular we are interested in the asymptotic behaviour of functions describing resource use in terms of some measure of problem size. This behaviour is often used as a basis for comparison between methods. Usually we prefer those methods whose resource use grows slowly as a function of problem size. This means we should be able to solve larger problems faster. We will revisit these issues towards the end of this note.

Kinds of Problems

In the study of algorithms you will meet a wide diversity of kinds of algorithms. However there are a few classes of problem that arise in many different circumstances. In this note we introduce three classes of problem you will meet again and again in your study of algorithms. These are decision, counting and optimisation problems. The following sections consider each in a little more detail.

**Decision Problems**

Decision problems are problems that have yes/no answers. Here are a few examples:
1. Given a map of all the navigable waterways in the UK and two places in the UK, is it possible to sail between the two places or not?

2. Given a string of symbols, is it a syntactically correct Java program or not?

3. Given a propositional logic formula $\phi$, does it have a satisfying valuation or not?

4. Given two finite state acceptors, do they accept the same language?

In looking for methods to solve these problems, we are looking for some kind of measure of the “size” of a problem and how the time taken to carry out the method varies with the size of problem. For logic problems like the third one, we have various potential choices, but a useful one is the number of different propositional variables in the formula. If we are given a formula with $n$ different propositional variables in it, how long will it take for a solution method to give us an answer? If we choose the very simple method of checking all rows in the truth table, then we have to check $2^n$ rows for equivalence, and may need to check $2^n$ rows to check for satisfiability. Ideally, we would like to find a faster approach or at least a method that works faster under some circumstances. We will consider these problems further in CS1Bh.

Counting Problems

Another kind of problem that is closely related to decision problems are those related to counting the number of different solutions to a problem. For example:

1. Given a propositional formula, how many satisfying valuations does it have?

2. Given a map of all the navigable waterways in the UK and two places in the UK, how many (loop-free) ways are there to sail between the two places?

Counting problems are often harder to solve than the corresponding decision problem. For example, you might imagine there could be a fast way to check that a formula is satisfiable but it seems harder to find the count of all satisfying valuations without considering all the candidates.

Optimisation Problems

Optimisation problems involve finding the “best” solution to a problem according to some measure of what “best” is. For example:

1. Given a map of all the navigable waterways in the UK with how deep each waterway is and two places in the UK, find the deepest route connecting the two places. This would allow us to check that we could sail a particular boat that required a given depth of water between two places on the map.

2. Given a map of all the navigable waterways in the UK with the length of each section of waterway and two places in the UK, find the shortest route between the two places.
3. Given a map of all the navigable waterways in the UK with the length of each section of waterway. Find the shortest tour that visits all the places on the map. This might be useful if we were planning some kind of inspection trip to look at all the destinations.

4. Given a language, find a finite state acceptor with a smallest number of states that accepts it.

Optimisation problems are a very significant class of problems and have been extensively studied. On the face of it we might think they are even harder than counting problems because they require us to evaluate how good each potential solution is and compare it to the others.

**Problem Areas**

Most algorithms have their origins in a particular problem area but many algorithms have a wider area of application than their originating problem area. In CS1Bh we consider a few problem areas and consider some algorithms arising from the area. The next sections provide brief descriptions of the main areas we consider.

**Sorting and Searching**

We have already met some algorithms for sorting and searching in CS1Ah. These algorithms have a very wide area of application and we still spend some more time in CS1Bh considering efficient algorithms for these problems. Sorting is one of the few areas where we know our algorithms are as efficient as they possibly can be. In the area of sorting we have lower bound results that establish strong bounds on how fast any sorting algorithm can be. We also have sorting methods that achieve these lower bounds. Establishing lower bounds is very hard because we are proving a result that establishes that all possible algorithms must go slower than the lower bound. Such results are hard to establish and there are relatively few good lower bound results for particular problems.

**Graph algorithms**

Graphs are a pervasive data structure in computer science. There are hundreds of interesting computational problems defined in terms of graphs. For instance, the map of the navigable waterways in the UK (but also the map of London’s subway, or the structure of the Web, by interpreting links as edges between URLs) can be described as graphs.

**Logic**

The satisfiability problem consists of checking whether a particular propositional formula has a satisfying valuation. This is a very hard problem that can take considerable computational resource to check. However in many practical cases there are simple rules-of-thumb that can help us find satisfying valuations very quickly.
The satisfiability problem is particularly important because many other problems can be modelled as satisfiability problems in propositional logic. For instance, the functionality of two circuits, for instance a simple and a sophisticated adder of two 32-bit numbers, can be encoded as two formulas. If the two formulas have the same truth table, then we know that the sophisticated adder behaves indeed like an adder, and so that no error was made in its design. This \textit{equivalence} problem above can be reduced to the satisfiability problem, i.e., it is possible to check that $\phi_1$ and $\phi_2$ are equivalent (have the same truth table) by constructing a formula $\phi$ and checking if it is satisfiable. Can you see how $\phi$ should look like?

\textbf{Computational geometry}

Computational geometry studies algorithms for solving geometric problems. The input to a computational-geometry problem is a set of geometric objects, like points (given by their coordinates), line segments, or polygons. Typical decision problems are whether two lines intersect, or a point lies inside or outside a polygon. Other problems may require to compute an object, like the pair of closest points. Computational geometry finds applications in computer graphics, robotics, or VLSI design.

\textbf{The divide-and-conquer approach}

This approach to the design of algorithms is based on the idea that often very small instances of a problem have trivial solutions and that solutions to larger instances of the problem can be solved by \textit{dividing} the problem into a number of smaller problems, solving each of these and then recombing the solutions to these smaller problems to find a solution to the original problem. The Mergesort algorithm of Lecture Note 28 in CS1Ah is a good example:

- \textbf{Divide}: split the sequence of length $n$ into two halves of size $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$
- \textbf{Conquer}: sort the two subsequences recursively
- \textbf{Combine}: Merge the two sorted subsequences into a sorted sequence.

Divide-and-conquer algorithms in which the input can be split into two halves tend to be very efficient, but they are rare. More frequently, a problem of size $n$ can be solved by solving one or more problems of size $n - 1$ and combining the solutions.\footnote{In some books, ‘divide-and-conquer’ is reserved for the ‘split-in-two-halves’ approach.}

Algorithms designed using a divide-and-conquer approach are recursive in structure, and so the simplest way to implement them is usually by means of methods that recursively call themselves. However, recursive implementations can often be extremely inefficient, because they may perform the same computation many times. Very often, one starts with a recursive view of the algorithm, and then moves to a more efficient implementation, in which the recursive structure may no longer be visible.

The running time of recursive algorithms can often be described by a \textbf{recurrence equation} or \textbf{recurrence} which describes the overall running time of the algorithm in terms of the running time on smaller inputs.
We derive such a recurrence for Mergesort. Let \( T(n) \) be the running time of the algorithm on inputs of size \( n \). If \( n = 1 \), then we need \( \Theta(1) \) operations to sort the sequence. If \( n > 1 \), then we have to split the input. So we get

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1 \\
T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + \Theta(n) & \text{if } n > 1
\end{cases}
\]

In this way, determining the running time of an algorithm is reduced to solving a particular class of equation. Techniques to solve these equations are seen in CS3.

During the Computer Science course you will meet a wide range of solution techniques that can be deployed in writing algorithms. In the next lectures we introduce two of them.

**Greedy algorithms**

In an optimization problem we have a set of possible solutions, and our task consists of finding one solution that maximizes some parameter. For instance, in the first of the optimisation problems above, the parameter is the depth of the waterway.

Many algorithms for optimisation problems go through a sequence of steps, with a set of choices at each step. Greedy algorithms always make the choice that looks best at the moment. The hope is that a sequence of locally optimal choices will lead to an optimal solution. For some problems this hope becomes real, but for others it does not. We see examples of both.

**Dynamic Programming**

Dynamic programming is an approach that can often help to transform inefficient divide and conquer solutions into much more efficient methods. The basic approach of dynamic programming is to identify intermediate results that are recalculated in an inefficient method and store them so they need not be recalculated but are immediately available. We will see how this approach can greatly improve the efficiency of divide and conquer methods.

*Stuart Anderson and Javier Esparza, 27th January 2003.*