School of Computing and Information Systems
COMP90038 Algorithms and Complexity Tutorial Week 2

Sample Answers

The exercises

1. Consider the usual (unsigned) binary representation of integers. For example, `10110010` represents 178, and `000011` represents 3.

   (a) If we call the bits in an n-bit word $x_{n-1}, x_{n-2}, \ldots, x_2, x_1, x_0$ (so x_0 is the *least significant* bit), which natural number is denoted by $x_{n-1}x_{n-2}\cdots x_2x_1x_0$?

   (b) Describe, in English, an algorithm for converting from binary to decimal notation.

   (c) Write the algorithm in (pseudo-) code.

   (d) Describe, in English, how to convert the decimal representation to binary.

   Answer:

   (a) Assuming unsigned representation, n bits allows us to represent the integers from 0 to $2^n - 1$, inclusive. The bit-string $x_{n-1}x_{n-2}\cdots x_2x_1x_0$ denotes $\sum_{i=0}^{n-1} 2^i \cdot x_i$.

   (b) Here is one method, expressed in English. We build the decimal-notation number by visiting the binary digits from left to right, constructing the result in an "accumulator". Start with the accumulator being 0. As long as there is a next bit to process, double the value of the accumulator, and add the value of that next bit.

   (c) Notice how all sorts of ambiguities creep in when we use natural languages. You might easily get the impression that what was meant with the previous answer was "as long as there is a next bit, double the accumulator, and then, after all that doubling, add something." It isn't clear from the structure of the English sentence that "and add the value" is part of what should be done for each bit (and the use of a comma before "and" didn't help). In pseudo-code this should be made unambiguous:

   function BINTODEC($x_{n-1}x_{n-2}\cdots x_2x_1x_0$)
       $a \leftarrow 0$
       for $i \leftarrow 0$ to $n-1$ **do**
           $a \leftarrow 2a + x_{n-i-1}$
       return a

   (d) To convert decimal representation d to binary, the natural way is to generate the bits from right to left. To get the rightmost bit, calculate the parity of d, that is, find d mod 2. Then halve d (rounding down). Now repeat this process, to get the remaining bits. More precisely:

   function DECTOBIN(d)
       $n \leftarrow 0$
       while $d \neq 0$ **do**
           $x_n \leftarrow d \mod 2$
           $d \leftarrow \lfloor d/2 \rfloor$
           $n \leftarrow n + 1$
       return $x_{n-1}x_{n-2}\cdots x_2x_1x_0$

   This works for non-negative n.

2. Below are three (correct) formulas for calculating the area of a triangle with sides A, B, and C, of length a, b, and c, respectively.

   (a) $S = \sqrt{p(p-a)(p-b)(p-c)}$, where $p = (a+b+c)/2$
   (b) $S = \frac{1}{2}ab\sin\theta$, where θ is the angle between sides A and B
   (c) $S = \frac{1}{2}ah_A$, where h_A is the height to base A

   Which of these would you accept as an *algorithm*?

   Answer:

   (a) This is a fine algorithm, as long as the square root operation is a primitive operation available on our computing device, or we know how to calculate square roots. And actually, that's not too difficult. Here is an algorithm for finding the square root of a non-negative integer:

   function $\textsc{Sqrt}(n)$
       root $\leftarrow 1$
       while *root* $\cdot$ *root* $\leq n$ **do**
           root $\leftarrow$ *root* $+ 1$
       return *root* $- 1$

   Does this square root algorithm really work? As always in this subject, you should be skeptical and not accept an algorithm until you have tried it out on several examples. And even if it passes your tests you should still be somewhat skeptical, because there are infinitely many inputs and testing can only cover finitely many cases. In this example it is perhaps not so hard to see that the algorithm works for all cases.
   Here is an alternative algorithm which avoids the non-linear expression *root* $\cdot$ *root* (perhaps because general multiplication is expensive on the machine we use):

   function $\textsc{Sqrt}(n)$
       square $\leftarrow 1$
       $i \leftarrow 1$
       while *square* $\leq n$ **do**
           square $\leftarrow$ *square* $+ 2i + 1$
           $i \leftarrow i + 1$
       return $i - 1$

   Is this version correct? It tries to utilise the fact that $\sum_{i=1}^{k}(2i-1) = k^2$. And why would that be right?

   (b) This is a problematic formulation, because, even if the sine function is considered a primitive, there is no indication of how to compute the angle θ.

   (c) Again, the formula says "the height to base A", without any indication of how to find that. So we can't really call that an algorithm.

3. Consider the following problem: You are to design an algorithm to determine the best route for a subway passenger to take from one station to another in a city such as Kolkata or Tokyo.

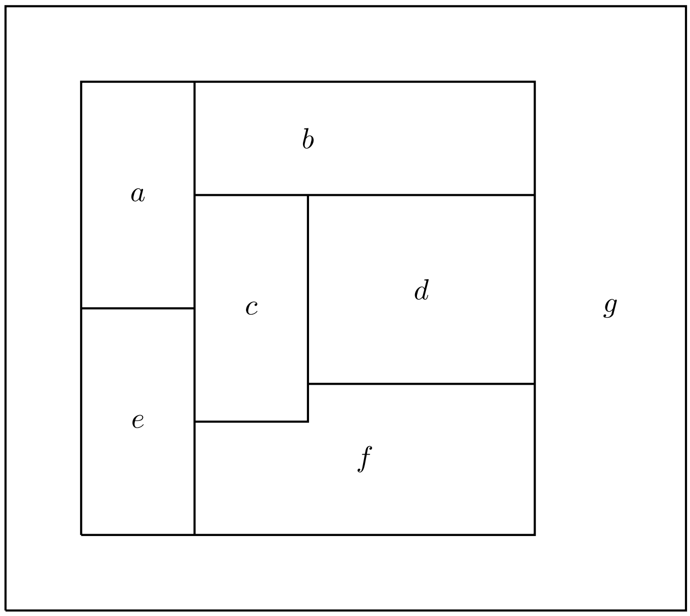
   (a) Discuss ways of making the problem statement less vague. In particular, what is "best" supposed to mean?
   (b) How would you model this problem by a graph?

   Answer:

   (a) In this context, "best" can mean many things. We may want to minimize the travel time, the number of train stops, the number of train changes, or some combination of these.

(b) The natural choice is to let nodes correspond to stations. Then there is an edge between two nodes iff the stations that correspond to the incident nodes are directly connected by a train line. If travel time is important (part of the definition of "best" route) then we need a weighted graph. In this case, we may also need to indicate how long it takes to change train, noting that a station may be on several lines. That information could be kept separately, or as annotations to stations, or we could do what some subway maps do: have several nodes for the same station.

4. Consider the following map:



(a) A cartographer wants to colour the map so that no two neighbouring countries have the same colour. How few colours can she get away with?

(b) Show how to reduce the problem to a graph-colouring problem.

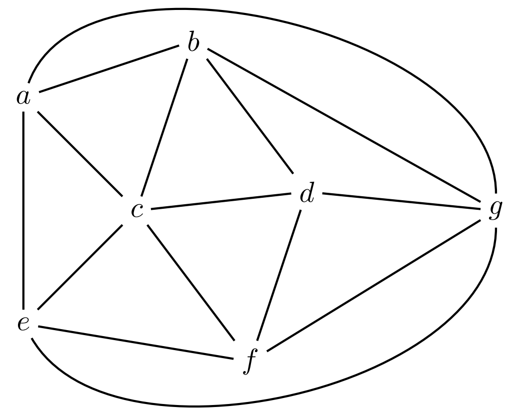
Answer:

(a) It turns out that four colours suffice for *any* planar map, no matter how complicated the map. Of course, for some maps fewer colours may be enough. The one given here does require four.

The result about the four colours has an interesting history. It has been conjectured to hold since 1852, and many incorrect proofs of the theorem have been given. Natural approaches to a proof involve extensive case analysis, too many cases to go through by hand. In 1976, Appel and Haken from the University of Illinois at Urbana-Champaign completed a proof by programming a computer to handle most of the tedious case analysis. At the time, there was much debate about the validity of such a proof: If it is too long for anybody to follow by reading, is it really a proof? Who says the program they used worked correctly—surely we also need a proof of *its* correctness. Since then, many independent computer-assisted proofs have been produced, and there is now a general consensus that the so-called four-colour theorem holds.

It is easy to determine whether a map can be coloured with one, two, or four colours (in the last case, the decision procedure can say 'yes' without even looking at its input). However, the case of three colours appears to be really hard. Technically it is "NP-complete", a concept we will discuss towards the end of this course.

(b) We can generate an undirected planar graph (*planar* meaning one that has no edges crossing), by placing a node in each of the "countries" a–g and connecting two nodes iff the corresponding countries have a common border. This is a general construction; it works for all maps. Now the question "how many colours are needed for the map?" becomes "how many colours are needed to colour the nodes of the graph, so that no two neighboring nodes have the same colour?" For our example, the graph looks like this:



5. You have to search for a given number n in a *sorted* list of numbers.

   (a) How can you take advantage of knowing that the list is represented as a linked (and sorted) list?

   (b) How can you take advantage of knowing the list is represented as an array?

   Answer:

   (a) We can stop searching as soon as we find (n or) a number greater than n.

   (b) With an array we can use binary search.

6. In the first lecture we discussed different ways of calculating the greatest common divisor of two positive integers. A mathematician might simply write

$$gcd(x, y) = max\{z \mid \exists u, v : x = uz \wedge y = vz\}$$

and suggest we develop a functional programming language that allows us to write this, leaving it to the language implementation to translate this definition to an efficient algorithm. Do you imagine a time when we may be able to do this? If we restrict our attention to functions like *gcd* which takes a pair of integers and returns an integer, do you think we may some day be able to automatically turn any function definition into a working algorithm?

Answer: The point of asking this question is to call attention to that fact that there are functions that are not *computable*. For some natural and precise meaning of "algorithm" (which we shall not discuss here), a non-computable function is one that cannot be captured by an algorithm. The (computable) function *gcd* has type $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers, and $\mathbb{N} \times \mathbb{N}$ is the set of pairs of natural numbers. Even if we restrict our interest to functions of that type, it turns out that there are vastly more functions than there are algorithms. (For those interested in the mathematics of this, there can be no more algorithms than natural numbers—discuss. And there is no surjective function from $\mathbb{N}$ to $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, let alone a bijection, by an argument known as "diagonalisation".)

It would be nice if we could say that non-computable functions are of academic interest only, because they are all weird functions with no practical application. However, that is not the case. Many natural, and important, functions cannot be captured by an algorithm.