Chapter 1. Section 3

Algorithms

Definition: algorithm = well-defined sequence of operations which, given any valid input value, will produce a valid output value.

This concept dates back at least to the 9th century when Islamic scholars such as al-Khwarizmi (the word 'algorithm' is a corruption of his name) devised systematic methodologies for trigonometry and algebra.

The modern theory of algorithms was developed in the 2nd half of the 20th century, giving an accepted formal description of what it means to calculate mathematically. Very roughly, anything we can compute (or can be computed by a computer) can be achieved in terms of 'basic instructions' (storing numbers, retrieving stored numbers and adding numbers is sufficient) provided we can:

1. execute instructions *in sequence*;
2. execute a sequence of instructions *conditionally* according to whether a Boolean variable evaluates to True or False;
3. *repeat* a sequence of instructions until some Boolean variable evaluates to False.

So the whole of computation reduces to **sequence**, **condition** and **repetition** applied to some basic set of instructions. In practice, we allow our basic instruction set to be much more generous than storage retrieval and addition, otherwise designing complicated algorithms for, say, graph theory problems, would be terribly complicated. Specifically, algorithms can be expressed as **pseudocode** which is an informal version of a programming language like Java or C, with English-language sentences added in *wherever it is clear how these can be turned into more precise instructions*. Our version of pseudocode is based on the mathematical programming language MAPLE.

An algorithm in pseudocode consists of a series of instructions or *statements*, generally each written on a separate line. Execution of the algorithm consists of performing the statements in turn, from top to bottom.

	statement	**syntax**	**example**
1	assignment	:=	$x := x + 1$;
2	comment	# blah blah	# this step increments x by 1
3	return	**return**(value)	**return**(x); # terminates execution
4	conditional	**if** test **then** instruction **end if**	**if** $x < y$ **then** $x := x + 1$ **end if**;

5		**if** test **then** instruction 1 **else** instruction 2 **end if**	**if** d(v) = 0 **then**
			delete v from the graph
			else
			delete edge from v;

6	loop	**for** list **do** instruction **end do**	**for** $v \in V$ **do** calculate d(v) **end do**;
7		**while** test **do** instruction **end do**	**while** d(v) ≥ 1 **do** remove some edge incident with v;

Note: the limit or 'scope' of the instructions in **if**, **for** and **while** statements is indicated by the words **end if** and **end do**, as appropriate. In practice it is acceptable, and often more clear, to use indentation to show this scope, as in examples 5 and 7.

The aim of this module is to discover or calculate properties of graphs by specifying correct and efficient algorithms. This process always involves the following three steps:

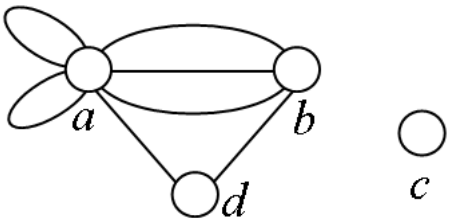
Specification: of the algorithm in pseudocode, starting with the name of the algorithm and its input and output;

Correctness: proof that the algorithm produces the correct output for every possible input;

Complexity: an analysis of how many steps the algorithm takes in order to produce the output for a given input (the technical name for this is the **complexity** of the algorithm).

Example 1

Degree sum We are given the task of calculating the sum of the degrees of a given graph. So given the graph



in which $\mathrm{d}(a) = 8$; $\mathrm{d}(b) = 4$; $\mathrm{d}(c) = 0$ and $\mathrm{d}(d) = 2$, the algorithm should return the output value of $\sum_{v \in V} \mathrm{d}(v) = 8 + 4 + 0 + 2 = 14$.

However, the input to our algorithm will **not** be the above picture! It will be a list of vertices (the set V) and a list of edges (set E). This is what our algorithm has to deal with. For the above graph we will have:

$$\begin{aligned} V &= \{a, b, c, d\}, \\ E &= \Big\{\{a\}, \{a\},\ \{a, b\},\ \{a, b\},\ \{a, b\},\ \{a, d\},\ \{b, d\}\Big\} \end{aligned}$$

We will give two algorithms, one called Dumb and one called Smart. The names indicate that our first algorithm takes a long time to accomplish its simple task!

Algorithm **Dumb**
Input: graph $G = (V, E)$
Output: $\sum_v \mathrm{d}(v)$

```
    degreesum:=0              # initialise the calculation
    for v ∈ V do
1.      degreesum := degreesum+ d(v)
    return(degreesum)
end
```

This is not quite a complete specification because we have not specified how to calculate $\mathrm{d}(v)$ at line 1. A full specification will replace line 1 with the following:

```
        d(v) := 0
2.      for e ∈ E do
            if v ∈ e then d(v) := d(v) + 1
        degreesum := degreesum+ d(v)
```

Proof of Correctness: it is obviously correct if there are no self-loops. The **if** statement in the loop at line 2 is **not** quite correct if e is a self-loop. But this is easy to correct.

Exercise: amend the specification of the calculation of $\mathrm{d}(v)$ to make it correct when the input graph can include self-loops.

Complexity: The **for** loop at line 1 takes $|V|$ steps since it makes 1 update of degreesum for each v in V.

For *each* of these $|V|$ steps the **for** loop at line 2 takes $|E|$ steps since we must check each edge e to see if v is incident with e.

So the total complexity is $|V| \times |E|$ steps.

We can produce an algorithm which uses fewer steps to achieve the same output!

Algorithm **Smart**
Input: graph $G = (V, E)$
Output: $\sum_v \mathrm{d}(v)$
    $|E| := 0$ # initialise the calculation
    for $e \in E$ **do** # loop to calculate $|E|$
        $|E| := |E| + |e|$
    return$(2 \times |E|)$
end

Proof of Correctness: this time it is not at all obvious that the output is going to be correct! Indeed it is quite mysterious — we ignore the vertices completely and just count edges! But there is a cute little lemma which says this is OK:

Lemma (The Handshaking Lemma) Theorem of the Day

For any graph $G = (V, E)$ we have

$$\sum_{v \in V} \mathrm{d}(v) = 2|E|.$$

Proof: Let us define a function f on pairs of vertices and edges:

$$f(v, e) = \begin{cases} 2 & \text{if } v \text{ and } e \text{ are incident and } e \text{ is a self-loop} \\ 1 & \text{if } v \text{ and } e \text{ are incident and } e \text{ is not a self-loop} \\ 0 & \text{if } v \text{ and } e \text{ are not incident} \end{cases}$$

Now for a given vertex v, $\sum_{e \in E} f(v, e) = \mathrm{d}(v)$. And for a given edge e, $\sum_{v \in V} f(v, e) = 2$. We count $f(v, e)$ over *all* vertices and edges. We can count by vertices and then edges or vice versa and get the same result:

$$\begin{aligned} \sum_{v \in V} \sum_{e \in E} f(v, e) &= \sum_{e \in E} \sum_{v \in V} f(v, e) \\ \text{so} \quad \sum_{v \in V} \mathrm{d}(v) &= \sum_{e \in E} 2 = 2|E|. \end{aligned}$$

□

Exercise: prove the Handshaking Lemma by induction on number of vertices.

Complexity: For algorithm Smart we have just the **for** loop which counts edges. This needs $|E|$ steps, to count through set E.

So algorithm Smart is faster than algorithm Dumb by a factor of $|V|$ — e.g. if G has 1000 vertices then Smart is 1000 times quicker than Dumb!

Example 2

Equal Degrees We are given the task of checking if a simple graph G (that is, no loops or multiple edges) has two vertices whose degrees are equal.

Here is a basic algorithm:

Algorithm **Stupid**
Input: simple graph $G = (V, E)$
Output: True if there are vertices $u, v \in V$ for which $d(u) = d(v)$; False otherwise
Found:=False # initialise the result
1. **for** $u \in V$ **do**
2. **for** $v \in V$ **do**
3. **if** $d(u) = d(v)$ **then** Found:=True
return(Found)
end

Proof of Correctness: obvious. We haven't said how $d(u)$ and $d(v)$ are to be calculated in step 3 but this was covered in our discussion of algorithm Dumb. And we do not have to worry about self-loops since our input graph G is simple.

Complexity: the **for** loop at 1. takes $|V|$ steps.

For each iteration of this loop the **for** loop at 2. takes $|V|$ steps. So that is $|V| \times |V|$ steps.

But for each of these steps we must calculate $d(u)$ and $d(v)$; and we saw that this requires $|E|$ steps for each degree. So the total number of steps is $|V|^2 \times 2|E|$.

We are not too bothered about the factor of 2: this is trivial compared to the factors of $|V|$ and $|E|$. In fact we have a special notation which allows us to forget about constant factors, or extra terms added on which are smaller: we write $O(|V|^2|E|)$ to mean $|V|^2|E|$ *up to constant multiples or the addition of smaller terms*. This is called 'big-Oh' notation and we shall describe it more formally soon.

Can we do better than an $O(|V|^2|E|)$ algorithm? We certainly can!

Algorithm **Clever**
Input: simple graph $G = (V, E)$
Output: True if there are vertices $u, v \in V$ for which $d(u) = d(v)$; False otherwise
if $|V| \geq 2$ **then return**(True) **else return**(False)
end

Proof of Correctness: More magic — this time we don't even count edges! Here is the proof:

Lemma If G is a simple graph on two or more vertices then G has two vertices with equal degree.

Proof: Let $n = |V|$. For every $v \in V$ we have $0 \leq d(v) \leq n - 1$. Moreover, we cannot achieve both bounds: the upper bound is achieved if some vertex is joined to all others, in which case no vertex can be isolated. So our n vertices take their degree values either from $\{0, \ldots, n-2\}$ or from $\{1, \ldots, n-1\}$. In either case we are taking n values from a set of size $n - 1$. By the Pigeon Hole Principle [1], some degree value must be given to at least 2 vertices. □

Complexity: we do not even have to calculate $|V|$: we just have to *start* counting vertices and stop when we get to 2. So the algorithm takes 2 steps at most. We write this as $O(1)$.

Exercise: is the above Lemma true for non-simple graphs? What about if self-loops are allowed but not multiple edges? What if multiple edges are allowed but not self-loops? Give counter examples as appropriate.

Exercise: We could speed up algorithm Stupid by only checking all $\binom{|V|}{2}$ pairs of vertices. The **for** loop at 1. checks each vertex but the **for** loop at 2. only checks vertices *further down the list*. What is the complexity of this new version of Stupid? Is it different from the original version when we use the big-Oh notation?

[1] The Pigeon Hole Principle says that if $n + 1$ balls are placed in n boxes then at least one box must contain more than one ball. Theorem of the Day