Lecture 2: Problems, Algorithms, Models of Computation and Programs

Aims:

- To understand, at a high level, what we mean by problem, algorithm, model of computation and program;
- To look at problems in more detail and understand what we mean by problem specification and problem instance.

2.1 Problems

- Here is an example of what we mean by the word 'problem' in this module.

Problem 2.1.
Parameters: *A positive integer, n.*
Returns: *The sum of the integers from 1 to n.*

- We are only interested in precisely-defined problems.

Here is another example.

Problem 2.2.
Parameters: *A positive integer, n.*
Returns: *YES if n is prime and NO if it isn't.*

Recall that a *prime number* is a positive integer ≥ 2 that can be divided without a remainder only by 1 and itself. For example, 2, 3, 5, 7, 11, 13, 17 and 19 are primes, whereas 4, 6, 8, 9, 10, 12, 14 and 15 are not. Non-primes are called *composite numbers*.

The following, on the other hand, is not precisely-defined:

Problem 2.3. *This is not what we would count as a problem!*
Parameters: *A topic, T.*
Returns: *An original poem on topic T.*

Here, it is not clear what we mean by a topic. "love" is a topic; is "3" a topic? "Happiness" is a topic; is "happy" a topic? It is not clear what we mean by a poem. Must it rhyme? Must it scan? According to whose speech patterns must it rhyme or scan? And it is not clear what we mean for a poem to be 'on' a topic. The following poem, for example, entitled *'Go North, South, East, and West, Young Man'*, is by Spike Milligan:

> Drake is going West, lads
> So Tom is going East
> But tiny Fred
> Just lies in bed,
> The lazy little beast.

What is its topic?

2.1.1 Problem Specifications

- A *problem specification* comprises
  - a characterisation of all legal inputs to the problem, and
  - a characterisation of the desired outputs as a function of the legal inputs.

Here's another example:

Problem 2.4.
Parameters: *Two non-negative integers, x and y.*
Returns: *The greatest common divisor of x and y.*

So some of the legal inputs are: $\langle 0, 16\rangle$, $\langle 80, 16\rangle$, $\langle 16, 80\rangle$, $\langle 80, 80\rangle$. The greatest common divisor (GCD) is also known as the highest common factor (HCF). It is the largest integer that exactly divides x and y. For example, the GCD of 80 and 32 is 16. 16 'goes into' 80 a whole number of times, and it goes into 32 a whole number of times; and there is no larger integer that goes into both a whole number of times.

2.1.2 Problems and Problem Instances

- In this problem,

Problem 2.4.
Parameters: *Two non-negative integers, x and y.*
Returns: *The greatest common divisor of x and y.*

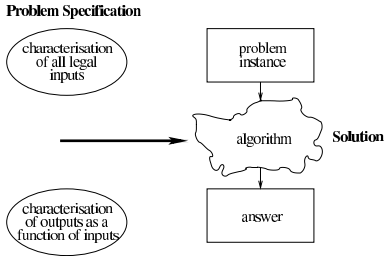
x and y are *formal parameters*.

They are just arbitrary, temporary names that allow us to unambiguously refer to the problem's inputs.

- Suppose we supply values,
  - e.g. we want to compute the GCD of 80 and 32.
  - 80 and 32 are the *actual parameters*.

- This is called *instantiating* the formal parameters.
- This gives us a *problem instance*.

- We speak of the *answer* to a *problem instance*:
  - e.g. the GCD of 80 and 32 is 16 (the answer);
  - e.g. the GCD of 24 and 80 is 8 (the answer).
- We speak of the *solution* to a *problem*:
  - the solution is an algorithm;
  - it tells us how to get the answer for *any* problem instance.



Class Exercise

Problem 2.2.
Parameters: *A positive integer, n.*
Returns: *YES if n is prime and NO if it isn't.*

1. Shout out some problem instances and answers.
2. Shout out an algorithm to solve the problem.

Class Exercise

How many instances do these problems have?

Problem 2.5.
Parameters: *An integer x where $0 \leq x \leq 5$.*
Returns: *YES if x is a member of the list $[2, 4, 6, 8, 10]$ and NO otherwise.*

Problem 2.6.
Parameters: *An integer x.*
Returns: *YES if x is a member of the list $[2, 4, 6, 8, 10]$ and NO otherwise.*

Problem 2.7.
Parameters: *A finite-length list of integers, L.*
Returns: *YES if 4 is a member of L and NO otherwise.*

Problem 2.8.
Parameters: *An integer x and a finite-length list of integers, L.*
Returns: *YES if x is a member of L and NO otherwise.*

Problem 2.9.
Parameters:
Returns: *YES if 4 is a member of the list $[2, 4, 6, 8, 10]$ and NO otherwise.*

Class Exercise

Shout out some algorithms for solving these two problems:

Problem 2.10.
Parameters: *An integer x and a finite-length list of integers, L.*
Returns: *42.*

Problem 2.11.
Parameters: *An integer x and a finite-length list of integers, L.*
Returns:

- Let's reflect on these different example problems.
- Problems 2.6, 2.7 and 2.8: These have an infinite number of problem instances. Be clear about this! Each input is finite. But there are an infinite number of them. Similarly, some problems may have an infinite number of possible outputs (e.g. problem 2.1). But again, be clear! The output for any particular instance is finite.
- Problem 2.5: Problems like this one, that have a finite set of instances, are not especially interesting. In principle, there is an easy way to solve such problems. We equip the algorithm with a table in which we have paired legal inputs with their answers. The algorithm simply searches the table to find the actual parameters and returns the corresponding answer. This approach cannot be taken if there is an infinite number of problem instances. Of course, it may also be impractical when there is a large but finite set of instances.
- Problem 2.9: Problems like this one, that have no parameters, are also not very interesting problems: they have only one instance and they will always produce the same result.
- Problems 2.10 and 2.11: Problems like these, which always produce the same answer for all instances, are not very interesting. They are easily solved!
- So the most interesting problems, from a Computer Science point of view, will be those with an infinite (or, at least, very large) number of instances, and ones in which the answer depends on the instance.

2.1.3 Decision Problems

- A *decision problem* is one whose return values are either YES or NO (or, equivalently, **true** or **false** or 0 or 1)

Problem 2.2.
Parameters: *A positive integer, n.*
Returns: *YES if n is prime and NO if it isn't.*

- These problems have been the focus of a lot attention in the theory of Computer Science.

2.1.4 Other Kinds of Problems

- There are other kinds of problems. But there is less agreement on how to classify them and what to call them. Here's one suggested classification:

  Search Problems: Find an x that satisfies property P.

  Enumeration Problems: Find all x that satisfy property P.

  Counting Problems: Count the number of xs that satisfy property P.

  Optimisation Problems: Find the x that best satisfies property P.

  Structuring Problems: Transform x to satisfy property P.

2.2 Algorithms

- OK, we're interested in precisely-defined problems whose solutions are algorithms. But what's an algorithm? That's a very deep question. We return to it much later in this module. For now, we'll try to obtain a rough idea. Let's start with some analogies:

'Algorithm'	Typical operations	Typical processor	Process
knitting pattern	knit one; purl one	little old lady	knitting a sweater
assembly instructions	glue panel A to strut B	model enthusiast	building a model plane
recipe	bring to the boil; stir in sugar	chef	making dessert
musical score	♫	pianist	playing a Beethoven sonata

- So, still refraining from a formal definition, here is a rough characterisation:

  An algorithm is

  - *a finite sequence of operations*
  - *for solving a problem.*
  - *Each operation is chosen from a set of well-defined operations.*

  If the algorithm is executed by a suitable processor on any instance of the problem, it will give rise to a process that

  - *halts in a finite time and*
  - *returns the answer for that problem instance.*

  Noteworthy aspects include the distinction between the text of the algorithm and the process that results when the operations in the text are executed. The text is finite. But the length of the process is not wholly determined by the length of the text: because of the possibility of iteration, certain operations may get executed multiple times. However, we insist that to qualify as an algorithm, the process must terminate.

  You might argue that requiring algorithms to terminate is undesirable. We may not always want termination. For example, you do not want your operating system (Linux or Windows) to terminate of its own accord; you want it to run forever (unless you intervene, e.g., by turning off the machine). Similarly, we don't want the software that monitors a nuclear power station to terminate of its own accord. Undoubtedly, these programs are important; they are the subject of research in theoretical Computer Science and software engineering. But they are not algorithms. For algorithms, we require termination.

- Let's look at an example of an algorithm.

Parameters: A positive integer, n.

Returns: The sum of the integers from 1 to n.

```
{   sum := 0;
    for i := 1 upto n
    {   sum := sum + i;
    }
    return sum;
}
```

- Here's another algorithm that solves the same problem.

Parameters: A positive integer, n.

Returns: The sum of the integers from 1 to n.

```
{   return n×(n+1)/2
}
```

This second algorithm probably seems better to you. Assuming both algorithms are correct (are they?), then the second looks more efficient. But, of course, the second is useless if our processor cannot handle multiplication and division. Or, suppose the processor can handle these operations but they execute many times more slowly than addition. Then the second algorithm may not be better for some or all problem instances. This begs the question of what our processor can do for us.

2.3 Models of Computation

- Our characterisation of algorithms mentions that its operations must be chosen from a set of operations. And, as we have just seen, this has a crucial bearing on the algorithms we can write and their relative efficiencies.
- We use the phrase *model of computation* to refer to the assumptions we make about the processor for which we are writing an algorithm.
- In particular, this determines the set of operations we may use.
- Our default model of computation will be
  - an errorless, sequential, digital computer;
  - we give instructions in an imperative-style; and
  - we have available the basic arithmetic and Boolean operations.

- We'll look in more detail at the way we allow ourselves to express algorithms (and hence the model of computation that we assume) in the next lecture.
- Note also that there will be occasions in the module when we consider alternative models of computation. For example, we might deliberately pretend that our model of computation does not include the ability to do multiplication. We can then see how we would solve problems in the absence of this operation. In other words, we would see how our algorithms differ when this operation is not available.

2.4 Programs

- An algorithm is a kind of abstract entity. It is independent of the language in which it is expressed and the processor which executes it.

  The main objective of the way we express algorithms in this module is ease of human comprehension.

  But, of course, if we want to arrange for a particular processor to execute an algorithm, then the algorithm needs to be expressed in some concrete form that can be understood by that processor.

  By analogy, Beethoven's sonata is an abstract entity that can be expressed in many concrete ways. But, if we express it in standard musical notation, then we can give it to certain processors: pianists! That is, we can give it to people who can read music and play the corresponding notes.

  With the kind of algorithms that we are interested in, there are again many concrete realisations. (To some extent, investigating these many concrete realisations is one of the topics of this module.) If the only intended processor is us, then a relatively informal concrete realisation will suffice, and we will develop a notation in the next lecture.
- If the processor is the CPU of some computer, then the algorithm must be given more precisely, as a *program*.
- We ask a *programmer* to *code* the algorithm using a *programming language*.
- Then, perhaps after some further translation to lower-level languages, the algorithm can be executed by the machine.
- There are many different programming languages and many different computer platforms. But the algorithm remains fundamentally the same, irrespective of which language and platform we choose.

Acknowledgements

The material in this lecture was influenced by Chapter 1 of [Raw91] and Chapter 2 of [Mor98]. The diagram comes from Harel's book [Har92]. Some of the example problems are taken from one of Harel's other books [Har00]. The table with the knitting pattern analogy is based on one in Chapter 1 of [GL82].

References

[GL82] L. Goldschlager and A. Lister. *Computer Science: A Modern Introduction*. Prentice-Hall, 1982.

[Har92] D. Harel. *Algorithmics: The Spirit of Computing*. Addison-Wesley, 2nd edition, 1992.

[Har00] D. Harel. *Computers Ltd.: What They really Can't Do*. Oxford University Press, 2000.

[Mor98] B. M. Moret. *The Theory of Computation*. Addison-Wesley, 1998.

[Raw91] G. J. E. Rawlins. *Compared to What? An Introduction to the Analysis of Algorithms*. W. H. Freeman, 1991.