specifies what the program should accomplish without describing how to do it. These notes will primarily be concerned with developing algorithms that map easily onto the imperative programming approach.

Algorithms can obviously be described in plain English, and we will sometimes do that. However, for computer scientists it is usually easier and clearer to use something that comes somewhere in between formatted English and computer program code, but is not runnable because certain details are omitted. This is called *pseudocode*, which comes in a variety of forms. Often these notes will present segments of pseudocode that are very similar to the languages we are mainly interested in, namely the overlap of *C* and *Java*, with the advantage that they can easily be inserted into runnable programs.

1.2 Fundamental questions about algorithms

Given an algorithm to solve a particular problem, we are naturally led to ask:

1. What is it supposed to do?
2. Does it really do what it is supposed to do?
3. How efficiently does it do it?

The technical terms normally used for these three aspects are:

1. Specification.
2. Verification.
3. Performance analysis.

The details of these three aspects will usually be rather problem dependent.

The *specification* should formalize the crucial details of the problem that the algorithm is intended to solve. Sometimes that will be based on a particular representation of the associated data, and sometimes it will be presented more abstractly. Typically, it will have to specify how the inputs and outputs of the algorithm are related, though there is no general requirement that the specification is complete or non-ambiguous.

For simple problems, it is often easy to see that a particular algorithm will always work, i.e. that it satisfies its specification. However, for more complicated specifications and/or algorithms, the fact that an algorithm satisfies its specification may not be obvious at all. In this case, we need to spend some effort *verifying* whether the algorithm is indeed correct. In general, testing on a few particular inputs can be enough to show that the algorithm is incorrect. However, since the number of different potential inputs for most algorithms is infinite in theory, and huge in practice, more than just testing on particular cases is needed to be sure that the algorithm satisfies its specification. We need *correctness proofs*. Although we will discuss proofs in these notes, and useful relevant ideas like *invariants*, we will usually only do so in a rather informal manner (though, of course, we will attempt to be rigorous). The reason is that we want to concentrate on the data structures and algorithms. Formal verification techniques are complex and will normally be left till after the basic ideas of these notes have been studied.

Finally, the *efficiency* or *performance* of an algorithm relates to the *resources* required by it, such as how quickly it will run, or how much computer memory it will use. This will