Chapter 2
Introduction to Problem Solving

They constructed ladders to reach to the top of the enemy's wall, and they did this by calculating the height of the wall from the number of layers of bricks at a point which was facing in their direction and had not been plastered. The layers were counted by a lot of people at the same time, and though some were likely to get the figure wrong the majority would get it right…Thus, guessing what the thickness of a single brick was, they calculated how long their ladder would have to be

Thucydides, The Peloponnesian War

'When I use a word,' Humpty Dumpty said, in a rather scornful tone, 'it means just what I choose it to mean — neither more nor less'
'The question is,' said Alice, 'whether you can make words mean so many different things'

Lewis Carroll, Through the Looking Glass and What Alice Found There

It is possible to invent a single machine which can be used to compute any computable sequence

Alan Turing

Aims
The aims of this chapter are:

- To examine some of the ideas and concepts involved in problem solving.
- To introduce the concept of an algorithm.
- To introduce two ways of approaching algorithmic problem solving.
- To introduce the ideas involved with systems analysis and design, i.e., to show the need for pencil and paper study before using a computer system.
- To introduce the Unified modelling Language - UML, a general purpose modelling language used in the field of software engineering.

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_2

2.1 Introduction

It is informative to consider some of the dictionary definitions of problem:

- A matter difficult of settlement or solution, Chambers.
- A question or puzzle propounded for solution, Chambers.
- A source of perplexity, Chambers.
- Doubtful or difficult question, Oxford.
- Proposition in which something has to be done, Oxford.
- A question raised for inquiry, consideration, or solution, Webster's.
- An intricate unsettled question, Webster's.

A common thread seems to be a question that we would like answered or solved. So one of the first things to consider in problem solving is how to pose the problem. This is often not as easy as is seems. Two of the most common methods to use here are:

- In natural language.
- In artificial or stylised language.

Both methods have their advantages and disadvantages.

2.2 Natural Language

Most people use natural language and are familiar with it, and the two most common forms are the written and spoken word. Consider the following language usage:

- The difference between a 3-year-old child and an adult describing the world.
- The difference between the way an engineer and a physicist would approach the design of a car engine.
- The difference between a manager and a worker considering the implications of the introduction of new technology.

Great care must be taken when using natural language to define a problem and a solution. It is possible that people use the same language to mean completely different things, and one must be aware of this when using natural language whilst problem solving.

Natural language can also be ambiguous: Old men and women eat cheese. Are both the men and women old?

2.3 Artificial Language

The two most common forms of artificial language are technical terminology and notations. Technical terminology generally includes both the use of new words and alternate use of existing words. Consider some of the concepts that are useful when examining the expansion of gases in both a theoretical and practical fashion:

- Temperature.
- Pressure.
- Mass.
- Isothermal expansion.
- Adiabatic expansion.

Now look at the following:

- A chef using a pressure cooker.
- A garage mechanic working on a car engine.
- A doctor monitoring blood pressure.
- An engineer designing a gas turbine.

Each has a particular problem to solve, and all will approach their problem in their own way; thus they will each use the same terminology in slightly different ways.

2.3.1 Notations

Some examples of notations are:

- Algebra.
- Calculus.
- Logic.

All of the above have been used as notations for describing both problems and their solutions.

2.4 Resume

We therefore have two ways of describing problems and they both have a learning phase until we achieve sufficient understanding to use them effectively. Having arrived at a satisfactory problem statement we next have to consider how we get the solution. It is here that the power of the algorithmic approach becomes useful.

2.5 Algorithms

An algorithm is a sequence of steps that will solve part or all of a problem. One of the most easily understood examples of an algorithm is a recipe. Most people have done some cooking, if only making toast and boiling an egg.

A recipe is made up of two parts:

- A check list of things you need.
- The sequence or order of steps.

Problems can occur at both stages, e.g., finding out halfway through the recipe that you do not have an ingredient or utensil; finding out that one stage will take an hour when the rest will be ready in ten minutes. Note that certain things can be done in any order — it may not make any difference if you prepare the potatoes before the carrots.

There are two ways of approaching problem solving when using a computer. They both involve algorithms, but are very different from one another. They are called top-down and bottom up.

The name algorithm is derived from the name of a ninth century Persian mathematician Abu Ja'far Mohammed ibn Musa al-Kuwarizmi (father of Ja'far Mohammed, son of Moses, native of Kuwarizmi), and has been corrupted in western culture as Al-Kuwarizmi.

2.5.1 Top-Down

In a top-down approach the problem is first specified at a high or general level: prepare a meal. It is then refined until each step in the solution is explicit and in the correct sequence, e.g., peel and slice the onions, then brown in a frying pan before adding the beef. One drawback to this approach is that it is very difficult to teach to beginners because they rarely have any idea of what primitive tools they have at their disposal. Another drawback is that they often get the sequencing wrong, e.g., now place in a moderately hot oven is frustrating because you may not have lit the oven (sequencing problem) and secondly because you may have no idea how hot moderately hot really is. However, as more and more problems are tackled, top-down becomes one of the most effective methods for programming.

2.5.2 Bottom-Up

Bottom-up is the reverse to top-down! As before you start by defining the problem at a high level, e.g., prepare a meal. However, now there is an examination of what tools, etc. you have available to solve the problem. This method lends itself to teaching since a repertoire of tools can be built up and more complicated problems can be tackled. Thinking back to the recipe there is not much point in trying to cook a six course meal if the only thing that you can do is boil an egg and open a tin of beans. The bottom-up approach thus has advantages for the beginner. However, there may be a problem when no suitable tool is available. A colleague and friend of the authors learned how to make Bechamel sauce, and was so pleased by his success that every other meal had a course with a Bechamel sauce. Try it on your eggs one morning. Here is a case of specifying a problem, prepare a meal, and using an inappropriate but plausible tool, Bechamel sauce.

The effort involved in tackling a realistic problem, introducing the constructs as and when they are needed and solving it is considerable. This approach may not lead to a reasonably comprehensive coverage of the language, or be particularly useful from a teaching point of view. case studies do have great value, but it helps if you know the elementary rules before you start on them. Imagine learning French by studying Balzac, before you even look at a French grammar book. You can learn this way but even when you have finished, you may not be able to speak to a Frenchman and be understood. A good example of the case study approach is given in the book Software Tools, by Kernighan and Plauger.

In this book our aim is to gradually introduce more and more tools until you know enough to approach the problem using the top-down method, and also realise from time to time that it will be necessary to develop some new tools.

2.5.3 Stepwise Refinement

Both of the above techniques can be combined with what is called stepwise refinement. The original ideas behind this approach are well expressed in a paper by Wirth, entitled "Program Development by Stepwise Refinement", published in 1971. It means that you start with a global problem statement and break the problem down in stages, into smaller and smaller sub problems that become more and more amenable to solution. When you first start programming the problems you can solve are quite simple, but as your experience grows you will find that you can handle more complex problems.

When you think of the way that you solve problems you will probably realise that unless the problem is so simple that you can answer it straight-away some thinking and pencil and paper work are required. An example that some may be familiar with is in practical work in a scientific discipline, where coming unprepared to the situation can be very frustrating and unrewarding. It is therefore appropriate to look at ways of doing analysis and design before using a computer.

2.6 Modular Programming

As the problems we try solving become more complex we need to look at ways of managing the construction of programs that comprise many parts. Modula 2 was one of the first languages to support this methodology and we will look at modular programming in more depth in a subsequent chapter.

2.7 Object Oriented Programming

There is a class of problems that are best solved by the treatment of the components of these problems as objects. We will look at the concepts involved in object oriented programming and object oriented languages in the next chapter.

2.8 Systems Analysis and Design

When one starts programming it is generally not apparent that one needs a methodology to follow to become successful as a programmer. This is usually because the problems are reasonably simple, and it is not necessary to be explicit about all of the stages one has gone through in arriving at a solution. As the problems become more complex it is necessary to become more rigorous and thorough in one's approach, to keep control in the face of the increasing complexity and to avoid making mistakes. It is then that the benefit of systems analysis and design becomes obvious. Broadly we have the following stages in systems analysis and design:

- Problem definition.
- Feasibility study and fact finding.
- Analysis.
- Initial system design.
- Detailed design.
- Implementation.
- Evaluation.
- Maintenance.

and each problem we address will entail slightly different time spent in each of these stages. Let us look at each stage in more detail.

2.8.1 Problem Definition

Here we are interested in defining what the problem really is. We should aim at providing some restriction on both the scope of the problem, and the objectives we set ourselves. We can use the methods mentioned earlier to help us out. It is essential that the objectives are:

- Clearly defined.
- Understood and agreed to by all people concerned, when more than one person is involved.
- Realistic.

2.8.2 Feasibility Study and Fact Finding

Here we look to see if there is a feasible solution. We would try and estimate the cost of solving the problem and see if the investment was warranted by the benefits, i.e., cost-benefit analysis.

2.8.3 Analysis

Here we look at what must be done to solve the problem. Note that we are interested in finding out what we need to do, but that we do not actually do it at this stage.

2.8.4 Design

Once the analysis is complete we know what must be done, and we can proceed to the design. We may find there are several alternatives, and we thus examine alternate ways in which the problem can be solved. It is here that we use the techniques of top-down and bottom-up problem solving, combined with stepwise refinement to generate an algorithm to solve the problem. We are now moving from the logical to the physical side of the solution. This stage ends with a choice among several alternatives. Note that there is generally not one ideal solution, but several, each with its own advantages and disadvantages.

2.8.5 Detailed Design

Here we move from the general to the specific, The end result of this stage should be a specification that is sufficiently tightly defined to generate actual program code.

It is at this stage that it is useful to generate pseudocode. This means writing out in detail the actions we want carried out at each stage of our overall algorithm. We gradually expand each stage (stepwise refinement) until it becomes Fortran — or whatever language we want.

2.8.6 Implementation

It is at this stage that we actually use a computer system to create the program(s) that will solve the problem. It is here that we actually need to know enough about a programming language to use it effectively to solve our problem. This is only one stage in the overall process, and mistakes at any of the stages can create serious difficulties.

2.8.7 Evaluation and Testing

Here we try to see if the program(s) we have produced will actually do what they are supposed to. We need to have data sets that enable us to say with confidence that the program really does work. This may not be an easy task, as quite often we only have numeric methods to solve the problem, which is why we are using the computer in the first place — hence we are relying on the computer to provide the proof; i.e., we have to use a computer to determine the veracity of the programs — and as Heller says, Catch 22.

2.8.8 Maintenance

It is rare that a program is run once and never used again. This means that there will be an ongoing task of maintaining the program, generally to make it work with different versions of the operating system or compiler, and to incorporate new features not included in the original design. It often seems odd when one starts programming that a program will need maintenance, as we are reluctant to regard a program in the same way as a mechanical object like a car that will eventually fall apart through use. Thus maintenance means keeping the program working at some tolerable level, often with a high level of investment in manpower and resources. Research in this area has shown that anything up to 80% of the manpower investment in a program can be in maintenance.

2.9 Unified Modelling Language - UML

UML is a general purpose modelling language used in the field of software engineering. It was developed by Grady Booch, Ivar Jacobson and James Rumbaugh whilst working at Rational Software in the 1990's. They were three of the leading exponents of object oriented software methodologies at the time and decided to unify the various approaches that each had developed.

UML combines techniques from data modelling (entity relationship diagrams), business modelling (work flows), object modelling, and component modelling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies.

It tends to be used more in business computing than scientific computing.

2.10 Conclusions

A drawback, inherent in all approaches to programming and to problem solving in general, is the assumption that a solution is indeed possible. There are problems which are simply insoluble — not only problems like balancing a national budget, weather forecasting for a year, or predicting which radioactive atom will decay, but also problems which are apparently computationally solvable.

Knuth gives the example of a chess problem — determining whether the game is a forced victory for white. Although there is an algorithm to achieve this, it requires an inordinately long time to complete. For practical purposes it is unsolvable.

Other problems can be shown mathematically to be undecidable. The work of Gödel in this area has been of enormous importance, and the bibliography contains a number of references for the more inquisitive and mathematically orientated reader. The Hofstader coverage is the easiest, and least mathematical.

As far as possible we will restrict ourselves to solvable problems, like learning a programming language.

Within the formal world of Computer Science our description of an algorithm would be considered a little lax. For our introductory needs it is sufficient, but a more rigorous approach is given by Hopcroft and Ullman in Introduction to Automata Theory, Languages and Computation, and by Beckman in Mathematical Foundations of programming.

2.11 Problems

2.1 What is an algorithm?

2.2 What distinguishes top-down from bottom-up approaches to problem solving? Illustrate your answer with reference to the problem of a car, motor-cycle or bicycle having a flat tire.

2.12 Bibliography

A.V. Aho A.V., Hopcroft J.E.,and J.D. Ullman J.D., The Design and Analysis of Computer Algorithms, Addison-Wesley, 1982.

- Theoretical coverage of the design and analysis of computer algorithms.

Beckman F.S., Mathematical Foundations of Programming, Addison-Wesley, 1981.

- Good clear coverage of the theoretical basis of computing.

Bulloff J.J., Holyoke T.C., Hahn S.W., Foundations of Mathematics — Symposium Papers Commemorating the 60th Birthday of Kurt Gödel, Springer-Verlag, 1969.

- The comment by John von Neumann highlights the importance of Gödel's work,.. Kurt Gödel's achievement in modern logic is singular and monumental — indeed it is more than a monument, it is a landmark which will remain visible far in space and time. Whether anything comparable to it has occurred in the logic of modern times may be debated. In any case, the conceivable proxima are very, very few. The subject of logic has certainly changed its nature and possibilities with Gödel's achievement.

Dahl O.J., Dijkstra E.W., Hoare C.A.R., Structured programming, Academic Press, 1972.

- This is the seminal book on structured programming.

Davis M., Computability and Unsolvability, Dover, 1982.

- The book is an introduction to the theory of computability and noncomputability — the theory of recursive functions in mathematics. Not for the mathematically faint hearted!

Davis W.S., Systems Analysis and Design, Addison-Wesley, 1983.

- Good introduction to systems analysis and design, with a variety of case studies. Also looks at some of the tools available to the systems analyst.

Edmonds D., Eidinow J., Wittgensteins Poker, Faber and Faber, 2001.

- The subtitle of the book provides a better understanding of the content - 'The story of a 10 minute argument between two great philosophers', which took place on Friday 25 October 1946 at the Cambridge Moral Science Club. The title of Poppers paper was 'Are there Philosophical problems?'. Ludwig Wittgenstein and Bertrand Russell were in the audience. Well worth a read.
- Here is an extract of a quote from the Times Literary Supplement. *A succinctly composed, informative, wonderfully readable and often funny account of a single impassioned encounter between the great overbearing philosopher Ludwig Wittgenstein and the younger, less great but equally overbearing philosopher Karl Popper... reads like an inspired collaboration between Iris Murdoch and Monty Python.*

Fogelin R.J., Wittgenstein, Routledge and Kegan Paul, 1980.

- The book provides a gentle introduction to the work of the philosopher Wittgenstein, who examined some of the philosophical problems associated with logic and reason.

Gödel K., On Formally Undecidable Propositions of Principia Mathematica and Related Systems, Oliver and Boyd, 1962.

- An English translation of Gödel's original paper by Meltzer, with quite a lengthy introduction by R.B. Braithwaite, then Knightbridge Professor of Moral Philosophy at Cambridge University, England, and classified under philosophy at the library at King's, rather than mathematics.

Hofstadter D.,The Eternal Golden Braid, Harvester Press, 1979.

- A very readable coverage of paradox and contradiction in art, music and logic, looking at the work of Escher, Bach and Gödel, respectively.

Hopcroft J.E., Ullman J.D., Introduction to Automata Theory, Languages and Computation, Addison-Wesley, 1979.

- Coverage of the theoretical basis of computing.

Jacobson, Ivar, Grady Booch, James Rumbaugh, (1998). The Unified Software Development Process. Addison Wesley Longman. ISBN 0-201-57169-2.

- The original book on UML.

Kernighan B.W., Plauger P.J., Software Tools, Addison-Wesley, 1976.

- Interesting essays on the program development process, originally using a non-standard variant of Fortran. Also available using Pascal.

Knuth D.E., The Art of Computer Programming, Addison-Wesley,

- Vol 1. Fundamental Algorithms, 1974
- Vol 2. Semi-numerical Algorithms, 1978
- Vol 3. Sorting and Searching, 1972
  - Contains interesting insights into many aspects of algorithm design. Good source of specialist algorithms, and Knuth writes with obvious and infectious enthusiasm (and erudition).

Millington D., Systems Analysis and Design for Computer Applications, Ellis Horwood, 1981.

- Short and readable introduction to systems analysis and design.

Popper K., The Logic of Scientific Discovery, 1934 (as Logik der Forschung, English translation 1959), Routledge, ISBN 0-415-27844-9.

- Popper argues that science should adopt a methodology based on falsifiability, because no number of experiments can ever prove a theory, but a single experiment can contradict one. A classic.

Salmon M.H., Logic and Critical Thinking, Harcourt Brace Jovanovich, 1984.

- Quite a good introduction to logic and critical thinking. Coverage of arguments, deductive and inductive arguments, causal arguments, probability and inductive logic, confirmation of hypotheses.

Wirth N., Algorithms + Data Structures = Programs, Prentice Hall, 1976.

- One of the seminal texts in computer science. Essential reading.

Wirth N., Program Development by Stepwise Refinement, Communications of the ACM, April 1971, Volume 14, Number 4, pp. 221–227.

- Clear and simple exposition of the ideas of stepwise refinement.