How to solve it by computer: Python edition

June 8, 2021

CONTENTS

Chapter 1

Introduction To Computer Problem-Solving

1.1 Introduction

Computer problem-solving can be summed up in one word- it is *demanding*! It is an intricate process requiring much thought, careful planning, logical precision, persistence, and attention to detail. At the same time it can be a challenging, exciting, and satisfying experience with considerable room for personal creativity and expression. If computer problem-solving is approached in this spirit then the chances of success are greatly amplified. In the discussion which follows in this introductory chapter we will attempt to lay the foundations for our study of computer problem-solving.

1.1.1 Programs and algorithms

The vehicle for the computer solution to a problem is a set of explicit and unambiguous instructions expressed in a programming language. This set of instructions is called a *program*. A program may also be thought of as an algorithm expressed in a programming language. An *algorithm* therefore corresponds to a solution to a problem that is *independent* of any programming language.

To obtain the computer solution to a problem once we have to program we usually have to supply the program with *input* or data. The program then takes this input and manipulates it according to its instructions and eventually produces an *output* which represents the computer solution to the problem. The realization of the computer output is but the last step in a very long chain of events that have led up to the computer solution to the problem.

Our goal in this work is to study in depth the process of algorithm design with particular emphasis on the problem-solving aspects of the task. There are many definitions of an algorithm. The following definition is appropriate in computing science. An *algorithm* consists of a set of explicit and unambiguous

finite steps which, when carried out for a given set of initial conditions, produce the corresponding output and terminate in a finite time.

1.1.2 Requirements for solving problems by computer

From time to time in our everyday activities, we employ algorithms to solve problems. For example, to look up someone's telephone number in a telephone directory we need to employ an algorithm. Tasks such as this are usually performed automatically without any thought to the complex under-lying mechanism needed to effectively conduct the search. It therefore comes as somewhat of a surprise to us when developing computer algorithms that the solution must be specified with such logical precision and in such detail. After studying even a small sample of computer problems it soon becomes obvious that the conscious *depth of understanding* needed to design effective computer algorithms is far greater than we are likely to encounter in almost any other problem-solving situation.

Let us reflect for a moment on the telephone directory look-up problem. A telephone directory quite often contains hundreds of thousands of names and telephone numbers yet we have little trouble finding the desired telephone number we are seeking. We might therefore ask why do we have so little difficulty with a problem of seemingly great size? The answer is simple. We quite naturally take advantage of the order in the directory to quickly eliminate large sections of the list and home in on the desired name and number. We would never contemplate looking up the telephone number of J. R. Nash by starting at page 1 and examining each name in turn until we finally come to Nash's name and telephone number. Nor are we likely to contemplate looking up the name of the person whose number is 2987533. To conduct such a search, there is no way in which we can take advantage of the order in the directory and so we are faced with the prospect of doing a number-by-number search starting at page 1. If, on the other hand, we had a list of telephone numbers and names ordered by telephone number rather than name, the task would be straightforward. What these examples serve to emphasize is the important influence of the data organization on the performance of algorithms. Only when a data structure is symbiotically linked with an algorithm can we expect high performance. Before considering these and other aspects of algorithm design we need to address the topic of problem-solving in some detail.

1.2 THE PROBLEM-SOLVING ASPECT

It is widely recognized that problem-solving is a creative process which largely defies systematization and mechanization. This may not sound very encouraging to the would-be problem-solver. To balance this, most people, during their schooling, acquire at least a modest set of problem-solving skills which they may or may not be aware of.

Even if one is not naturally skilled at problem-solving there are a number

of steps that can be taken to raise the level of one's performance. It is not implied or intended that the suggestions in what follows are in any way a recipe for problem-solving. The plain fact of the matter is that there is no universal method. Different strategies appear to work for different people.

Within this context, then, where can we begin to say anything useful about computer problem-solving? We must start from the premise that computer problem-solving is about understanding.

1.2.1 Problem definition phase

Success in solving any problem is only possible after we have made the effort to come to terms with or understand the problem at hand. We cannot hope to make useful progress in solving a problem until we fully understand what it is we are trying to solve. This preliminary investigation may be thought of as the *problem definition phase*. In other words, what we must do during this phase is work out *what must be done* rather *than how to do it*. That is, we must try to extract from the problem statement (which is often quite imprecise and maybe even ambiguous) a set of precisely defined tasks. Inexperienced problem-solvers too often gallop ahead with how they are going to solve the problem only to find that they are either solving the wrong problem or they are solving just a very special case of what is actually required. In short, a lot of care should be taken in working out precisely what must be done. The development of algorithms for finding the square root (algorithm 3.1) and the greatest common divisor (algorithm 3.3) are good illustrations of how important it is to carefully define the problem. Then, from the definitions, we are led in a natural way to algorithm designs for these two problems.

1.2.2 Getting started on a problem

There are many ways to solve most problems and also many solutions to most problems. This situation does not make the job of problem-solving easy. When confronted with many possible lines of attack it is usually difficult to recognize quickly which paths are likely to be fruitless and which may be productive.

Perhaps the more common situation for people just starting to come to grips with the computer solution to problems is that they just do not have any idea where to start on the problem, even after the problem definition phase. When confronted with this situation, what can we do? A block often occurs at this point because people become concerned with details of the implementation *before* they have completely understood or worked out an implementation-independent solution. The best advice here is not to be too concerned about detail. That can come later when the complexity of the problem as a whole has been brought under control. The old computer proverb' which says "the sooner you start coding your program the longer it is going to take" is usually painfully true.

1.2.3 The use of specific examples

A useful strategy when we are stuck is to use some props or heuristics (i.e. rules of thumb) to try to get a start with the problem. An approach that often allows us to make a start on a problem is to pick a specific example of the general problem we wish to solve and try to work out the mechanism that will allow us to solve this particular problem (e.g. if you want to find the maximum in a set of numbers, choose a particular set of numbers and work out the mechanism for finding the maximum in this set—see for example algorithm 4.3). It is usually much easier to work out the details of a solution to a specific problem because the relationship between the mechanism and the particular problem is more clearly defined. Furthermore, a specific problem often forces us to focus on details that are not so apparent when the problem is considered abstractly. Geometrical or schematic diagrams representing certain aspects of the problem can be usefully employed in many instances (see, for example, algorithm 3.3).

This approach of focusing on a particular problem can often give us the foothold we need for making a start on the solution to the general problem. The method should, however, not be abused. It is very easy to fall into the trap of thinking that the solution to a specific problem or a specific class-of problems is also a solution to the general problem. Sometimes this happens but we should always be very wary of making such an assumption.

Ideally, the specifications for our particular problem need to be examined very carefully to try to establish whether or not the proposed algorithm can meet those requirements. If the full specifications are difficult to formulate sometimes a well-chosen set of test cases can give us a degree of confidence in the generality of our solution. However, nothing less than a complete proof of correctness of our algorithm is entirely satisfactory. We will discuss this matter in more detail a little later.

1.2.4 Similarities among problem

We have already seen that one way to make a start on a problem is by considering a specific example. Another thing that we should always try to do is bring as much past experience as possible to bear on the current problem. In this respect it is important to see if there are any similarities between the current problem and other problems that we have solved or we have seen solved. Once we have had a little experience in computer problem-solving it is unlikely that a new problem will be completely divorced from other problems we have seen. A good habit therefore is to always make an effort to be aware of the similarities among problems. The more experience one has the more tools and techniques one can bring to bear in tackling a given problem. The contribution of experience to our ability to solve problems is not always helpful. In fact, sometimes it blocks us from discovering a desirable or better solution to a problem. A classic case of experience blocking progress was Einstein's discovery of relativity. For a considerable time before Einstein made his discovery the scientists of the day had the necessary facts that could have led them to relativity but it is almost certain

that their experience blocked them from even contemplating such a proposal-Newton's theory was correct and that was all there was to it! On this point it is therefore probably best to place only cautious reliance on past experience. In trying to get a better solution to a problem, sometimes too much study of the existing solution or a similar problem forces us down the same reasoning path (which may not be the best) and to the same dead end. In trying to get a better solution to a problem, it is usually wise, in the first instance at least, to try to *independently* solve the problem. We then give ourselves a better chance of not falling into the same traps as our predecessors. In the final analysis, every problem must be considered on its merits.

A skill that it is important to try to develop in problem-solving is the ability to view a problem from a variety of angles. One must be able to metaphorically turn a problem upside down, inside out, sideways, back-wards, forwards and so on. Once one has developed this skill it should be possible to get started on any problem.

1.2.5 Working backwards from the solution

There are still other things we can try when we do not know where to start on a problem. We can, for example, in some cases assume that we already have the solution to the problem and then try to work backwards to the starting conditions. Even a guess at the solution to the problem may be enough to give us a foothold to start on the problem. (See, for example, the square root problem-algorithm 3. 1). Whatever attempts that we make to get started on a problem we should write down as we go along the various steps and explorations made. This can be important in allowing us to systematize our investigations and avoid duplication of effort. Another practice that helps us develop our problem-solving skills is, once we have solved a problem, to consciously reflect back on the way we went about discovering the solution. This can help us significantly. The most crucial thing of all in developing problem-solving skills is practice. Piaget summed this up very nicely with the statement that "we learn most when we have to invent."

1.2.6 General problem-solving strategies

There are a number of general and powerful computational strategies that are repeatedly used in various guises in computing science. Often it is possible to phrase a problem in terms of one of these strategies and achieve very considerable gains in computational efficiency.

Probably the most widely known and most often used of these principles the *divide-and-conquer* strategy. The basic idea with divide-and-conquer is to divide the original problem into two or more sub-problems which can hopefully be solved more efficiently by the same technique. If it is possible to proceed with this splitting into smaller and smaller sub-problems we will eventually reach the stage where the sub-problems are small enough to be solved without further splitting. This way of breaking down the solution to a problem has found wide

application in particular with sorting, selection, and searching algorithms. We will see later in Chapter 5 when we consider the binary search algorithm how applying this strategy to an ordered data set results in an algorithm that needs to make only $\log_2 n$ rather than n comparisons to locate a given item in an ordered list n elements long. When this principle is used in sorting algorithms, the number of comparisons can be reduced from the order of n^2 steps to $n \log_2 n$ steps, a substantial gain particularly for large n. The same idea can be applied to file comparison and in many other instances to give substantial gains in computational efficiency. It is not absolutely necessary for divide-and-conquer to always exactly halve the problem size. The algorithm used in Chapter 4 to find the k^{th} smallest element repeatedly reduces the size of the problem. Although it does not divide the problem in half at each step it has very good average performance.

It is also possible to apply the divide-and-conquer strategy essentially in reverse to some problems. The resulting *binary doubling strategy* can give the same sort of gains in computational efficiency. We will consider in Chapter 3 how this complementary technique can be used to advantage to raise a number to a large power and to calculate the n^{th} Fibonacci number. With this doubling strategy we need to express the next term n to be computed in terms of the current term which is usually a function of n/2 in order to avoid the need to generate intermediate terms.

Another general problem-solving strategy that we will briefly consider is that of *dynamic programming*. This method is used most often when we have to build up a solution to a problem via a sequence of intermediate steps.The monotone sub-sequence problem in Chapter 4 uses a variant of the dynamic programming method. This method relies on the idea that a good solution to a large problem can sometimes be built up from good or optimal solutions to smaller problems. This type of strategy is particularly relevant for many optimization problems that one frequently encounters in operations research. The techniques of *greedy search*, *backtracking* and *branch-and!-bound* evaluations are all variations on the basic dynamic programming idea. They all tend to guide a computation in such a way that the minimum amount of effort is expended on exploring solutions that have been established to be sub-optimal. There are still other general computational strategies that we could consider but because they are usually associated with more advanced algorithms we will not proceed further in this direction.

1.3 TOP-DOWN-DESIGN

The primary goal in computer problem-solving is an algorithm which is capable of being implemented as a correct and efficient computer program. In our discussion leading up to the consideration of algorithm design we have been mostly concerned with the very broad aspects of problem-solving. We now need to consider those aspects of problem-solving and algorithm design which are closer to the algorithm implementation.

Once we have defined the problem to be solved and we have at least a vague idea of how to solve it, we can begin to bring to bear powerful techniques for designing algorithms. The key to being able to successfully design algorithms leis in being able to manage the inherent complexity of most problems that require computer solution. People as problem-solvers are only able to focus on, and comprehend at one time, a very limited span of logic or instructions. A technique for algorithm design that tries to accommodate this human limitation is known as *top-down design* or *step-wise refinement.*

Top-down design is a strategy that we can apply to take the solution of a computer problem from a vague outline to a precisely defined algorithm and program implementation. Top-down design provides us with a way of handling the inherent logical complexity and detail frequently encountered in computer algorithms. It allows us to build our solutions to a problem in a step-wise fashion. In this way, specific and complex details of the implementation are encountered only at the stage when we have done sufficient groundwork on the overall structure and relationships among the various parts of the problem.