5 Algorithmics

As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise – by what course of calculation can these results be arrived at by the machine in the shortest time?
Charles Babbage[1]

Beginnings

What is an algorithm? The word is derived from the name of the Persian scholar Mohammad Al-Khowarizmi (see B.5.1 and Fig. 5.1). In the introduction to his classic book *Algorithmics: The Spirit of Computing*, computer scientist David Harel gives the following definition:

> An algorithm is an abstract recipe, prescribing a process that might be carried out by a human, by a computer, or by other means. It thus represents a very general concept, with numerous applications. Its principal interest and use, however, is in those cases where the process is to be carried out by a computer.[2]

Thus an algorithm can be regarded as a "recipe" detailing the mathematical steps to follow to do a particular task. This could be a numerical algorithm for solving a differential equation or an algorithm for completing a more abstract task, such as sorting a list of items according to some specified property. The word *algorithmics* was introduced by J. F. Traub in a textbook in 1964 and popularized as a key field of study in computer science by Donald

B.5.1. The word algorithm derives from the name of a ninth-century mathematician, Mohammad Al-Khowarizmi. He was a Persian scholar who studied in the House of Wisdom, a library and research center in Baghdad. Al-Khowarizmi wrote an early text on the rules for carrying out mathematical operations with Hindu-Arabic numbers – the numbers we still use today. This book, in its Latin translation *Algoritmi de numero Indorum*, was very influential in introducing the use of Hindu-Arabic numerals and the positional representation of numbers throughout Europe. In Latin, al-Khowarizmi became known as Algoritmi, from which we get the word *algorithm*. It is also from him that we get the word *algebra* – from the Latin title of another of his books.

Knuth (B.5.2) and David Harel (B.5.3). When the steps to define an algorithm to carry out a particular task have been identified, the programmer chooses a programming language to express the algorithm in a form that the computer can understand.

Fig. 5.1. A page from al-Khowarizmi's book on algebra. In this book he describes the steps for solving linear and quadratic equations and lays down the foundations of algebra as a new discipline of mathematics. The original meaning of the word *algebra* in Arabic is "to restore" – this refers to balancing out both sides of an equation. It is hard to overstate the importance of algebra in mathematics.

The earliest known algorithm was invented between 400 and 300 B.C. by the Greek mathematician Euclid. Euclid's algorithm is a method for finding the greatest common divisor, or GCD, of two positive integers. For example, the fraction 8/12 can be reduced to 2/3 by dividing both numerator and denominator by their GCD, which in this case is 4. The algorithm to find the GCD of two numbers, M and N, can be expressed in four steps (see Fig. 5.2):

Step 1: Input values M, N.
Step 2: Divide M by N to find the remainder R.
Step 3: If R is zero then N is the answer, print N.
Step 4: If R is not zero, change the value of M to N and the value of N to R and go back to Step 2.

How does this algorithm work? Any number that divides both M and N must also divide the remainder R. Similarly, any number that divides both N and R must also divide M. This means that the GCD of M and N is the same as the GCD of N and R. We can see the algorithm in action in Table 5.1 for finding the GCD of 65 and 39. We begin by dividing 65 by 39 – we are interested only in the remainder, which is 26. We assign M the value of 39 and N the value of the remainder 26. In the next iteration we calculate the remainder again and assign values to M and N. We repeat the process until the remainder becomes zero, in this case the value of GCD will be held in variable N.

As Harel says in the title of his book, algorithms can be regarded as the "spirit of computing." They are the precise procedures required to get computers to do something useful. In this chapter we will look at some examples of different types of algorithms. Historically, computers were used to solve numerical problems so we will start by looking at algorithms for numerical simulations. We will also introduce the idea of using random numbers to derive approximate answers to complex simulations. These "Monte Carlo" methods were first used in the Manhattan atomic bomb project. We will then look at problems such as finding the quickest way to sort a list of names and finding the shortest path from one city to another – as we now do routinely with our global positioning system, or GPS, navigation systems. This will lead us to a discussion of the efficiency of algorithms and an introduction to computational complexity theory.

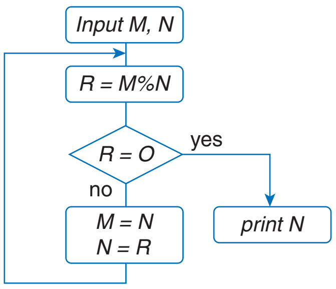


Fig. 5.2. Flowchart of the GCD algorithm. The "%" operation calculates the remainder when M is divided by N.

Table 5.1. Euclid's algorithm for GCD of 65 and 39

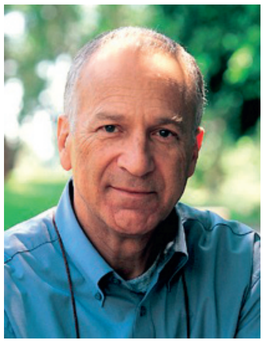
	M	N	R
Iteration 1	65	39	26
Iteration 2	39	26	13
Iteration 3	26	13	0
Result	GCD = 13		

B.5.2. Donald Knuth with his series of books called *The Art of Computer Programming* has made a major contribution to the cataloging and systematic analysis of algorithms. This book is generally accepted as the "gold standard" in the field. Knuth offered a prize of one hexadecimal dollar, that is, \$2.56, for each error found in his books. These checks are considered to be trophies in academic circles. While writing his book, Knuth also developed the TeX typesetting software that is still very widely used. After winning a programming competition in the 1960s, Knuth was asked how he managed it. He replied: "When I learned how to program, you were lucky if you got five minutes with the machine a day. If you wanted to get the program going, it just had to be written right. So people just learned to program like it was carving in stone. You have to sidle up to it. That's how I learned to program."[B1]

Numerical algorithms

In Chapter 1 we saw that the ENIAC computer was originally built to calculate the trajectories of shells for artillery tables. In mathematics, the solution to this trajectory problem is obtained by solving the differential equation that arises from an application of Newton's laws of motion. On a computer, such differential equations must be approximated using some numerical method.

Let's look at a simple example. Suppose we have an object falling under the force of gravity. To find the velocity of the object at any given time, we need to solve Newton's law for the rate of change of velocity with time. If we assume that the object is dropped from rest, we can calculate the velocity at any later time using Newton's law in the form of a differential equation. Mathematically we can treat the velocity and time values as varying "continuously." However, computers can only store individual "discrete" values of the velocities and time – such as the velocity after one second, the velocity after two seconds and so on. We need to approximate the differential equation by splitting up the time variable into very small increments. We can then calculate the approximate incremental change of velocity for each small increment of time. There are many different "numerical methods" that can be used to find such approximate solutions to differential equations on a computer. The simplest numerical approximation is a method devised by the Swiss mathematician Leonhard Euler (B.5.4). In practice, Euler's method is not very precise and there are more accurate numerical methods available to solve such differential equations. Figure 5.3 shows how Euler's method compares to the exact solution for the problem of finding the velocity of an object falling under gravity through a fluid, and subject to a resistance proportional to the square of the velocity. It was the FORTRAN programming language that first gave scientists the capability of writing their programs as a relatively straightforward translation of their mathematical equations, instead of having to program the solutions to these problems using low-level machine language or assembly language.


B.5.3. David Harel gave a series of lectures on Israeli radio in 1984 explaining computer algorithms to a general audience. This led to his famous book *Algorithmics: The Spirit of Computing* and to his more recent book on computability entitled *Computers Ltd: What they really can't do.*

Another important numerical technique for simulations goes by the name of the "Monte Carlo" method. In 1946, physicists at Los Alamos Laboratory were investigating the distance that neutrons can travel through various

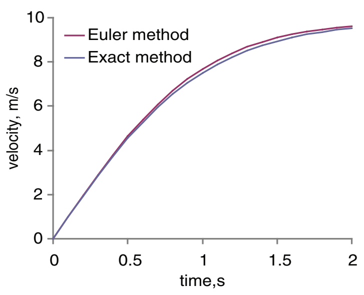


Fig. 5.3. Comparison of a numerical solution of a simple differential equation obtained using Euler's method with the exact analytical solution. If we use the Corrector-Predictor method, which is much more accurate than the simpler Euler method, we obtain results that are almost indistinguishable from the exact solution.

materials. Polish American mathematician Stanislaw Ulam (B.5.5), who, along with Edward Teller, is credited for devising a workable mechanism for the hydrogen bomb, came up with the idea of using many random experiments to find an approximate answer to the problem. He recalled his inspiration as follows:

> The first thoughts and attempts I made to practice [the Monte Carlo method] were suggested by a question which occurred to me in 1946 as I was convalescing from an illness and playing solitaires. The question was what are the chances that a Canfield solitaire laid out with 52 cards will come out successfully? After spending a lot of time trying to estimate them by pure combinatorial calculations, I wondered whether a more practical method than "abstract thinking" might not be to lay it out say one hundred times and simply observe and count the number of successful plays. This was already possible to envisage with the beginning of the new era of fast computers, and I immediately thought of problems of neutron diffusion and other questions of mathematical physics, and more generally how to change processes described by certain differential equations into an equivalent form interpretable as a succession of random operations. Later [in 1946], I described the idea to John von Neumann, and we began to plan actual calculations.[3]

Von Neumann chose the code name Monte Carlo for the new technique in reference to the famous casino where Ulam's uncle used to like to gamble. The first unclassified paper on Monte Carlo methods, authored by Nicholas Metropolis and Ulam, was published in 1949.

We can use the Monte Carlo method to find an approximate value of π in the following way. Imagine we place a dartboard inside a square as shown in Figure 5.4. If we throw darts randomly at the square, the number of darts that land within the circle is proportional to the area of the circle. By comparing the

B.5.4. Leonhard Euler (1707–83) was a Swiss mathematician and physicist. Euler was born in Basel, the son of Paul Euler, a pastor of the Reformed Church, and a friend of Johann Bernoulli, then Europe's foremost mathematician. Bernoulli recognized the young Euler's genius and convinced him to pursue mathematics rather than enter the church. At the age of thirteen, Leonhard was a student at the University of Basel and received his Master of Philosophy in 1723 for his dissertation that compared the philosophies of Descartes and Newton. After he was unable to secure a university position in Basel, in 1726 Euler was offered a post at the Russian Academy of Sciences in St. Petersburg. In 1741 he took up a position at the Berlin Academy and spent a very creative twenty-five years in Germany before returning to St. Petersburg.

Euler's mathematical abilities were supplemented by his having a photographic memory. He could recite Virgil's *Aeneid* from beginning to end and could even remember the first and last lines on each page of his edition of the book. Euler worked in almost all areas of mathematics – geometry, calculus, algebra, trigonometry, and number theory – as well as in physics and astronomy and he introduced and popularized many of the notational conventions in mathematics that we still use today. Euler was the first to write $f(x)$ to denote the function f applied to the argument x, as well as the modern notation for the trigonometric functions such as sine, cosine, and tangent; the letter e for the base of the natural logarithm; the Greek letter Σ for summations; and the letter i for complex numbers. Euler is also responsible for what physicist Richard Feynman called "the most remarkable formula in mathematics,"[B2] Euler's identity: $e^{i\pi} + 1 = 0$.

number of darts that land within the square to the number that land within the circle of radius R, we can obtain an estimate of π:

$$(\text{Number of darts inside circle})/(\text{Number of darts inside square}) = (\text{Area of circle})/(\text{Area of square}) = \pi R^2/4R^2 = \pi/4$$

In order for this method to give an accurate value of π, we need the darts to be thrown genuinely at random, so that they cover the entire area uniformly. We also need a large number of throws. Because generating large numbers of truly random numbers is extremely difficult, von Neumann developed a clever algorithm to generate "pseudorandom" numbers on the computer. Given an initial starting number as a "seed," these pseudorandom numbers are then generated deterministically by von Neumann's algorithm and approximate a truly random distribution. This technique has the advantage that the exact sequence of numbers can be reproduced by starting with the same seed, and this turns out to be very helpful in debugging Monte Carlo simulation programs.

Although we have only given a very simple example here, Monte Carlo methods can be used to evaluate complex integrals in a similar manner. These methods are now widely used in many areas of science and business – and in computer algorithms for playing games.

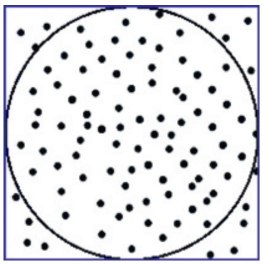


Fig. 5.4. Determination of π using Monte Carlo method. The diagram shows a circle of radius R inside a square of side 2R. The area of circle is πR^2 and the area of the square is $4R^2$. Throwing randomly distributed darts gives an estimate of π by comparing the number of darts landing inside the circle to the number landing inside the square.

Sorting

Although the earliest electronic computers were generally used to find numerical solutions to scientific problems, it was clear from early on that they were capable of solving other types of problems. During World War II, the Colossus computer at Bletchley Park in the United Kingdom was used for breaking codes, and soon after the war the LEO computer demonstrated the utility of computers for assisting with routine business problems, such as stock keeping, distribution, and payroll.

Let's take a look at how computers handle such nonnumerical tasks. We will do so by examining the problem of sorting a list of names into alphabetical order. We will show how this can be done using two different algorithms,

B.5.5. Stanislav Ulam was born in 1909 in the city of Lwow in Poland, now the city of Lviv in the Ukraine. He studied mathematics at university and was a member of the Lwow School of Mathematics. The members met at the Scottish Café in Lwow and recorded their discussions in the "Scottish Book." Ulam met John von Neumann and was invited to visit the institute at Princeton in 1935. He left Poland in 1939 just before the German invasion and many members of his family died in the Holocaust. In 1943 Ulam was an assistant professor at the University of Wisconsin in Madison and he asked von Neumann if he could join the war effort. As a result he received a letter from Hans Bethe inviting him to join the Manhattan Project, a top-secret project to build the atom bomb, based at Los Alamos, near Santa Fe, New Mexico. Because he knew nothing about New Mexico, Ulam checked out a guidebook from the university library. On the checkout slip he found the names of three colleagues who had mysteriously "disappeared" a few months before! At Los Alamos he worked on numerical solutions to the hydrodynamical equations for the plutonium implosion bomb. After the war, Ulam returned to Los Alamos to work on the development of the hydrogen bomb. In 1951, Ulam and Edward Teller came up with a mechanism for a working fusion bomb using "radiation implosion."

called *bubble sort* and *merge sort*. Suppose we have the following list of eight names that we want to sort alphabetically:

Bob
Ted
Alice
Pat
Joe
Fred
May
Eve

Letters are usually represented in a computer using the so-called ASCII scheme, an acronym for the American Standard Code for Information Interchange. All of the twenty-six Standard English characters, plus punctuation and other symbols can be represented as a seven-bit ASCII code. Hence we can arrange for the computer to understand what we mean when we ask for two numbers to be compared and placed in alphabetical order.

Let's first examine the bubble sort algorithm. It works by repeatedly comparing adjacent names and interchanging them if they are out of alphabetical order. We start by considering the bottom two names on the list:

May	*swap*	Eve
Eve		May

Next we move the "bubble" up and consider the next pair on the list:

Fred	*swap*	Eve
Eve		Fred

We repeat the process until the bubble of paired names has reached the top. We are then guaranteed that the correct first name is at the top of the list. The detailed workings of this first iteration are shown in Figure 5.5.

Now start again, with the bubble again at the bottom of the list. At the end of this second pass through all of the names on the list, the second name will be in the correct position, second from the top. For sorting all eight items correctly we need to repeat this process seven times. You can see why this algorithm is called the bubble sort, because the sorted names bubble up to the top.

The bubble sort algorithm gets the job done, but it is not a very efficient way to sort a large list. Sorting is such a common task that computer scientists have spent a lot of time looking for efficient sorting algorithms. One very clever and practical algorithm is called merge sort. It was invented by von Neumann

Fig. 5.5. An example of the bubble sort algorithm. This example works by exchanging adjacent names if they are out of order, starting from the bottom of the list, and continuing until the bubble of paired names has reached the top. This is repeated until all items are sorted correctly.

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
Bob	Bob	Bob	Bob	Bob	Bob	Bob	Alice
Ted	Ted	Ted	Ted	Ted	Ted	Alice	Bob
Alice	Alice	Alice	Alice	Alice	Alice	Ted	Ted
Pat	Pat	Pat	Pat	Eve	Eve	Eve	Eve
Joe	Joe	Joe	Eve	Pat	Pat	Pat	Pat
Fred	Fred	Eve	Joe	Joe	Joe	Joe	Joe
May	Eve	Fred	Fred	Fred	Fred	Fred	Fred
Eve	May	May	May	May	May	May	May

in 1945 and uses a fundamental technique of computer science called *divide-and-conquer*. We begin by splitting our eight-name list into two halves, so that we have two lists of four names. We then split each half again into two lists of two names. We order each of the pairs of names and then merge the sorted pairs. The merge is done by repeatedly comparing the characters at the head of each list and sending the alphabetically lower item to the output. We complete the sort by merging the two sorted lists of four names in the same way. The diagram in Figure 5.6 illustrates how the merge sort algorithm works.

In programming the merge sort algorithm, we can write the program using "recursion" for the divide phase, creating a subroutine calling itself a number of times. In this case, we can introduce a subroutine called "Divide" and use it to split the list into two halves. If there are only two elements left in the list, it returns them in alphabetical order; if there are more than two elements, the Divide subroutine calls itself and repeats the process. This carries on until there are only one or two elements left in the divided list. The use of recursion is a very powerful programming technique much loved by computer scientists. The merge phase can also be programmed recursively using a Merge subroutine. We will compare the efficiencies of some algorithms later in this chapter, in the section on complexity theory.

Graph problems

We are familiar with the use of routing algorithms from our GPS navigation systems. Indeed the systems are becoming so reliable that we are fast approaching a time when finding our way by reading a paper map will be a lost art! All we do to find the shortest route from A to B is to enter the start and end points in the car navigation system. How do computers solve such problems? The solution uses another branch of mathematics invented by Euler: "graph theory."

The city of Königsberg in Prussia – now Kaliningrad, Russia – was famous for a long-standing puzzle in mathematics. The city is located on both sides of the Pregel River, and there are two large islands in the river connected to the mainland by seven bridges (Fig. 5.7a). The seven bridges problem was to find a walk through the city that would cross each bridge only once. In 1735, Euler

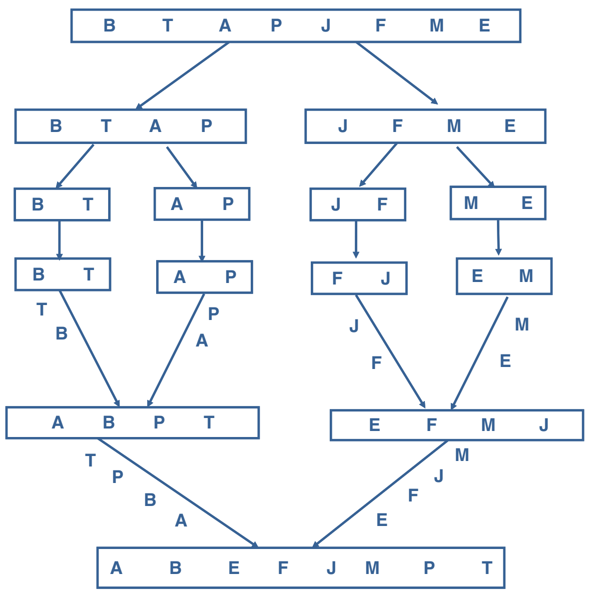


Fig. 5.6. An example of the merge sort algorithm that uses a divide-and-conquer approach to reduce the list to sets of pairs of names. These are ordered and the different pairs merged together in the correct order.

proved that there was no solution, and in so doing he laid the foundations of graph theory and the beginnings of the study of topology.

Euler solved the problem by reducing it to essentials. The choice of route on land is unimportant: only the sequence of bridges crossed is relevant (Fig. 5.7b). The map can be further simplified by replacing each landmass with a dot – called a "vertex" or a "node" – and each bridge by a line – called an "edge" – joining two vertices (Fig. 5.7c). Only the connection information in the resulting "graph" is important for this problem, not details of the layout of the figure. This illustrates one of the key ideas of topology: topology is not concerned with the rigid shape of objects or surfaces, just their connectivity.

Euler then observed that, except for the start and finish of the walk, whenever one enters a vertex (landmass) by a bridge, one must leave the same landmass or vertex by another bridge. If each bridge is crossed only once, except

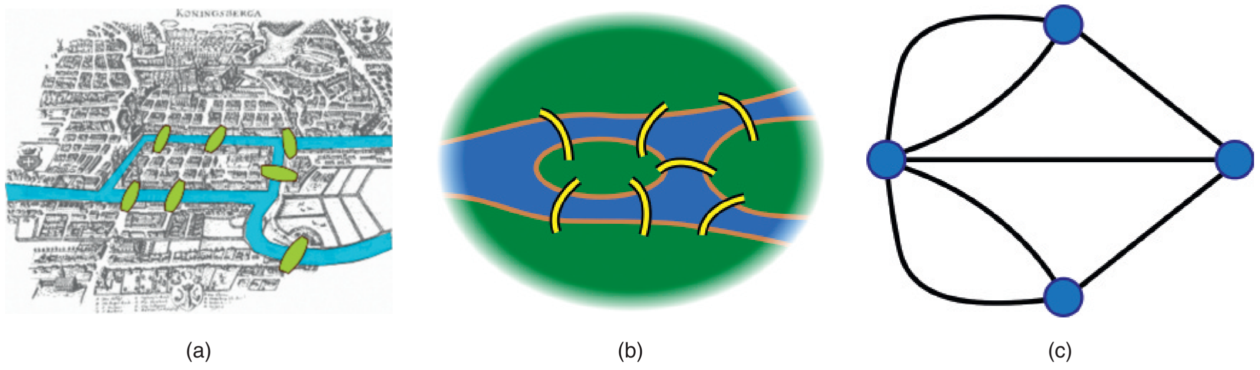


Fig. 5.7. Three representations of the Seven Bridges of Königsberg: (a) Königsberg in Euler's time; (b) a more abstract representation of the seven bridges; and (c) a graph of the seven bridges.

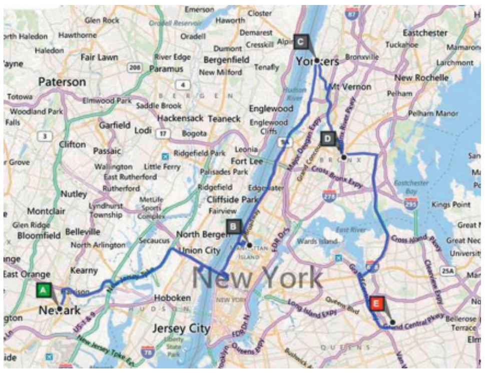
for the start and finish landmasses, the number of bridges connecting any other landmass must be an even number – half of the bridges for the walker to enter the landmass, and half for the walker to leave it. In the case of the bridges in Königsberg, we see that all the four landmasses are connected by an odd number of bridges – one by five, the other three by three. Because at most two of the landmasses can be the starting and end points, we see immediately that it is not possible to walk through the city crossing each bridge exactly once.

Let us look at another important type of graph problem. This is the problem of finding the minimal spanning tree (MST) – a path that reaches every node in a graph with the minimum cost. Consider five well-known communities in the area around New York City (Fig. 5.8a) and represent them as a graph (Fig. 5.8b). In this graph, each community is represented as a vertex, with a road joining two communities by an edge. Each edge is assigned a number representing the "cost" needed to go between the communities at the ends of each edge. This could represent the cost of a cable connection or the time of travel between the two places, for example.

Imagine that the company wants to connect its offices in the five communities using the least amount of optical fiber. The minimal spanning tree (MST) solves this problem. Finding the MST is a problem that can be solved by using a so-called greedy algorithm. Greedy algorithms take the optimal choice at each local stage of the algorithm and in general are not guaranteed to find the globally best solution but can be proved to do so for the case of the MST. In this example, we start with the shortest edge in the graph; then from the two vertices at the ends of this edge, we choose the next shortest edge. We continue to add to the resulting graph by adding the next shortest edge that has not yet been considered. We repeat this procedure until we have visited each city in the graph. The result is the MST shown in Figure 5.9.

The solution illustrates another important structure in computer science: trees. Trees are similar to graphs except that they do not contain closed loops. Trees are found everywhere in our daily lives, such as in the organization charts of companies or in the file structures on your computer (Fig. 5.10). Efficient algorithms to traverse and manipulate tree structures are an important area of algorithmics (Fig. 5.11).

(a)



(b)

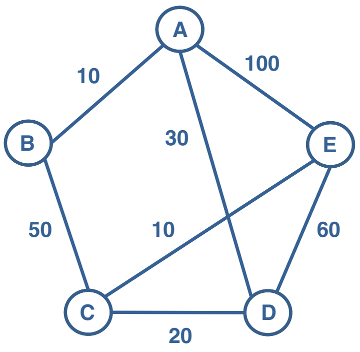


Fig. 5.8. An illustration of the MST problem. The figure shows (a) five communities in the New York area: A = Newark; B = Manhattan; C = Yonkers; D = The Bronx; E = Queens; and (b) a graph of the five communities with distances allocated to each edge.

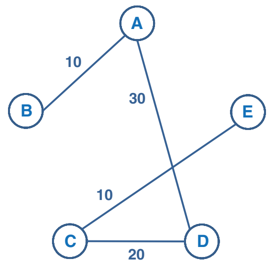


Fig. 5.9. The cheapest solution connecting all five cities with the minimum length of optical fiber is the MST for the graph in Figure 5.8.b. In this case the MST can be found using a simple greedy algorithm, as explained in the text.

Let us look at another important problem. This is the problem of finding the shortest path through a graph – the sort of algorithm used by our GPS navigation systems. How does the computer embedded in our GPS system solve such problems? It does so by using a variant of the shortest path algorithm devised by Edsger Dijkstra, an early computer science pioneer in the area of programming languages and software engineering. Dijkstra was asked in an interview how he came to invent his shortest path routing algorithm and he replied:

What is the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path which I designed in about 20 minutes. One morning I was shopping with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path.[4]

Let us go back to our company in the New York area. Suppose that the company's headquarters are located in Newark (city A) and that it frequently needs to deliver supplies from its headquarters to each of its offices located in the communities of Manhattan (B), Yonkers (C), the Bronx (D), and Queens (E). For simplicity's sake, let's assume that the edges of our graph are "directed" like one-way streets, meaning they can only be traveled in one direction. In addition, we need to ensure that there is no possibility of these directed edges forming a closed loop or cycle; with our one-way restrictions this is true of the graph in Figure 5.12. This type of graph occurs in many places in computer science and has the intimidating name of a Directed Acyclic Graph, or DAG.

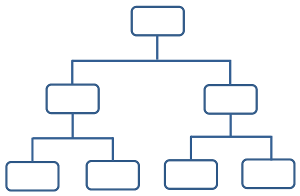


Fig. 5.10. An example of a tree structure organization. The tree data structure is one of the key concepts of computer science and they are the cornerstones of all databases. A tree consists of nodes and branches. Whenever a node is added or removed the tree needs to be adjusted in order to make it shorter and more "bushy" rather than tall and thin. This makes search operations much faster.

To find the shortest path from the head office in Newark to every other office, Dijkstra's algorithm uses a greedy method. Let us see how Dijkstra's algorithm works in this case:

- The first iteration of the algorithm starts at headquarters A and finds the office that has the shortest direct connection to A. In our example of Figure 5.8b, the closest office to A is clearly B, with the a distance of 10. (Note that because there is no direct connection from city A to office C, we set distance to infinity.)
- The next step in the algorithm examines the shortest paths to the other offices if we start from A as before, but also now allow the option of going through B. We see that by going through B, we can now get to C and therefore we record the distance as 10 + 50 = 60. For the next step in the algorithm we need to add to our set of two locations, A and B, the next closest office to A. The next shortest path is now to office D, with a distance of 30.
- For the third iteration we now allow paths from A that can either go directly from A or via locations B or D. With this extra option, we see by inspecting the graph (Fig. 5.8b) that the shortest path from A to C is now through D rather than through B. Similarly, it is now shorter to get to E through D than going direct from A. Again, we complete the step by looking for the city with the next shortest path from A, which is now C with a distance of 50.

Fig. 5.11. Cartoon of a self-adjusting tree.

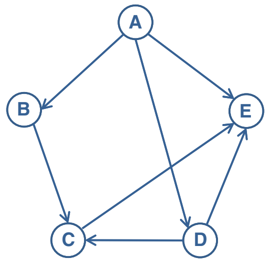


Fig. 5.12. An example of a Directed Acyclic Graph or DAG.

- For the next and final iteration, we calculate the shortest paths from A to each other location, but now allowing any of the offices B, D, and C as possible intermediate destinations. The shortest path to E is now through offices B and C rather than through D or going direct from A.

In this way, we have now found the shortest path from A to all the other locations in the graph. These iterations of Dijkstra's algorithm are summarized in Table 5.2.

There are many other types of routing algorithms that can be applied to such shortest path problems. One important method is called *dynamic programming*, which is a technique that can be used when simple greedy algorithms do not give the best solution. Dynamic programming algorithms allow for long-range optimizations instead of the purely local optimizations performed in Dijkstra's algorithm.

Before we leave this section, we want to introduce an important character in the study of algorithmics and graph problems – the traveling salesman problem, or TSP, which has fascinated mathematicians and computer scientists since the 1930s. The problem can be stated as follows: given a list of cities and the distances between each of them, what is the shortest route that a traveling salesman can take to visit each city and return to his starting point?

Obviously one way of solving this problem is just to use brute force and enumerate every possible route. For our five-location network in Figure 5.8b, we can calculate how many different routes the salesman could take. The problem is equivalent to finding the number of permutations of the five symbols A, B, C, D, and E. Because any shortest route starts and finishes at the same city, using any of the five cities as the starting point of the route gives the same answer. So we can just start with A and look for all the possible routes starting with A. There are then four possible choices for the second city, three for the third, and two for the fourth, before we are left with only the fifth city. Thus it looks like we need to evaluate $4 \times 3 \times 2 \times 1 = 24$ permutations (or 4!, to use the common notation for factorials). But there is another simplification. The distance from B to C is clearly the same as the distance from C to B, and the same is true for every pair of cities. Each permutation has a reverse permutation of the same length, and it does not matter which direction we travel round the tour. We therefore need to consider only $4!/2 = 12$ different routes.

The shortest path for this problem, ABECDA, is shown in Figure 5.13. Where is the difficulty with the TSP? For an N-city problem, we need to examine $(N - 1)!/2$ tours, and as the number of cities increases, this brute force

Table 5.2 Iterations of Dijkstra's algorithm. Column S is a set of cities used in the shortest path search. D[node] is the distance to a city

Iteration	S	D[B]	D[C]	D[D]	D[E]
Initial	{A}	10	∞	30	100
#1	{A,B}	10	60	30	100
#2	{A,B,D}	10	50	30	90
#3	{A,B,D,C}	10	50	30	70
#4	{A,B,D,C,E}	10	50	30	70

approach rapidly becomes impractical. We can therefore say that such a brute force algorithm for the N-city problem is *unreasonable*. To understand better what we mean by *reasonable* and *unreasonable*, we need to look at how we can measure the performance of algorithms. Before we do this, we will give a brief history of attempts at solving the TSP for large numbers of cities.

In 1954, three researchers at the RAND Corporation in Santa Monica, California – George Dantzig (B.5.6), Ray Fulkerson, and Selmer Johnson – looked at the problem of finding the shortest path for a tour through all the forty-eight contiguous U.S. states. *Newsweek* reported their success:

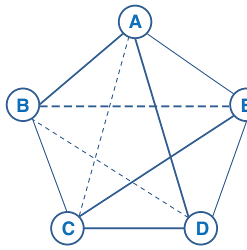
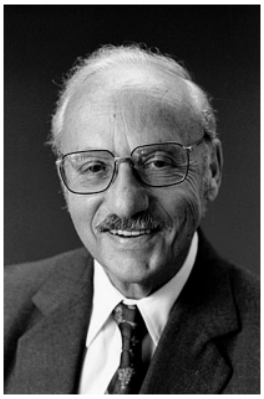


Fig. 5.13. The TSP for our five-city problem corresponds to finding the shortest round tour through all of the cities. Note that to conform to the usual formulation of the TSP problem with all-to-all paths possible between the cities, we have added in the missing "direct" paths between B and D, B and E, and A and C, taking the distances as the shortest distances to go between them, going via an intermediate node.

> Finding the shortest route for a traveling salesman – starting from a given city, visiting each of a series of other cities, and then returning to the original point of departure – is more than an after-dinner teaser. For years it has baffled not only goods- and salesman-routing businessmen but mathematicians as well. If a drummer visits 50 cities, for example, he has 10^{62} (62 zeros) possible itineraries. No electronic computer in existence could sort out such a large number of routes and find the shortest.
>
> Three RAND Corp. mathematicians, using Rand McNally distances between the District of Columbia and major cities in each of the 48 states, have finally produced a solution. By an ingenious application of linear programming – a mathematical tool recently used to solve production-scheduling problems – it took only a few weeks for the California experts to calculate "by hand" the shortest route to cover the 49 cities: 12,345 miles.[5]

The algorithm the three researchers used to solve the problem was unusual: it was just a board with pegs at the city locations and a piece of string to try out possible TSP tours. As the *Newsweek* blurb recounts, they found the shortest tour by using a powerful technique called *linear programming*. Dantzig had devised the technique as a method to schedule the training, supply, and deployment of military units when he was working at the Pentagon after World War II.

Linear programming expresses the problem as an economic model with inputs and outputs as variables subject to a set of constraints. These constraints can include inequalities, such as requiring some variables to always be greater than or equal to zero. As the name implies, the variables were combined in a set of linear equations and the goal was to choose the variables to maximize an explicit objective. To find the optimal solution to such a linear programming problem, Dantzig developed an algorithm that was named one of "The Top Ten Algorithms of the Century" in the year 2000. This is the simplex algorithm, which is still widely used in industry where the models can have hundreds of thousands of constraints and variables. A detailed discussion of this algorithm is beyond the scope of this book, but it still provides the basis for modern analyses of the TSP. Using linear programming, Dantzig, Fulkerson, and Johnson were able to prove that their solution was indeed the shortest path, writing:



B.5.6. George Dantzig (1914–2005) is credited with the development of the simplex algorithm and numerous other contributions to linear programming. His algorithm is used to solve many real-life problems related to air traffic scheduling, logistics, planning processes in oil refineries, circuit design, and many more.

> In this context, the tool of choice is linear programming, an amazingly effective method for combining a large number of simple rules, satisfied by all tours, to obtain a single rule of the form "no tour through this point set can be shorter than X." The number X gives an immediate quality measure: if we can also produce a tour of length X then we can be sure that it is optimal.[6]

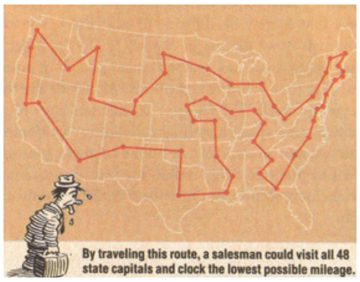

By traveling this route, a salesman could visit all 48 state capitals and clock the lowest possible mileage.

Fig. 5.14. Optimal tour around the United States visiting forty-eight state capitals. Researchers George Dantzig, Ray Fulkerson, and Selmer Johnson from the RAND Corporation did not actually use the forty-eight state capitals in their classic 1954 solution of the forty-eight-city problem.

Since the pioneering work of these RAND researchers, the challenge of the traveling salesman has continued to attract the attention of researchers. The record for finding the optimal tour has been steadily increased from 48 cities (Fig. 5.14) in 1954; to 64 by Michael Held and Richard Karp in 1971; to 532, then 1,002, and then 2,392 cities by teams led by Martin Grotschel and Manfred Padberg in 1987; to tours of 13,509 cities in the United States in 1998 and of 24,978 cities in Sweden in 2004 by Concorde, the current champion TSP program. The Concorde program was developed by David Applegate, Robert Bixby, Vasek Chvatal, and William Cook and is available over the Internet. In 2006, they used their program to find the shortest travel time for a laser to cut connections in a Bell Labs computer chip. The result was an optimal tour for an 85,900 "city" problem (Fig. 5.15). This stands as the record TSP for which the optimal tour is known. Larger problems, such as the 100,000-city Mona Lisa problem created by the artist Bob Bosch and shown in Figure 5.16, are significantly more difficult than the computer chip problem, which has many "cities" close together on straight lines. Currently the best solution for a Mona Lisa tour is still 0.0026 percent above the bound for the optimal tour!

Before we leave the traveling salesman problem we should say that although finding a provably optimal tour is still computationally challenging, there are many practical ways to find very good approximate solutions to the TSP. Most modern algorithms are variants on a method devised by Bell Labs researchers Shen Lin and Brian Kernighan in 1973. This systematizes the process of making incremental tour improvements on some initial tour. A "2-opt" move is an improvement wherein two edges are deleted and the tour reconnected with two shorter edges. Similarly, we can look for 3-opt moves and more. Danish computer scientist Keld Helsgaun improved on the original Lin-Kernighan method in 1998 by explicitly incorporating a search for 5-opt exchanges, reconnecting ten edges at a time. Combining Lin-Kernighan with ideas from simulated annealing in physics, in 1991 researchers Olivier Martin, Steve Otto, and Ed Felten at Caltech developed what is now known as the Chained Lin-Kernighan algorithm. In 2000, this method was used on a 25,000,000-city problem to find a tour that was only about 0.3 percent greater than the theoretical shortest path. This is still the dominant algorithm for use with very large data sets. The TSP is an important optimization problem for many types of problems – from various pickups and deliveries, to finding markers on genomes, to moving telescopes and manufacturing electronic circuit boards.

Complexity theory

As Charles Babbage foresaw in the quotation that introduces this chapter, now that we have computers, the question of how to find the fastest algorithm to solve a particular problem moves to center stage. In our discussion on sorting

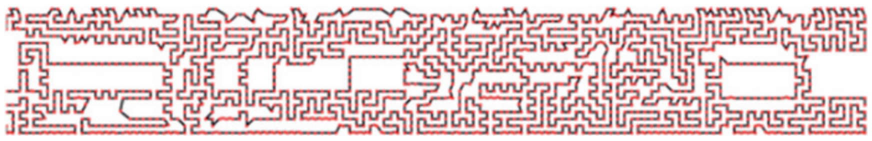


Fig. 5.15. Section of the optimal tour for the 85,900-"city" problem.

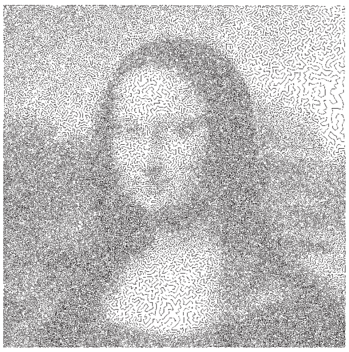


Fig. 5.16. In 2009, Robert Bosch from Oberlin College generated a set of 100,000 points and then ran the TSP algorithm on this set in order to calculate the minimal path. As the algorithm proceeds it connects the dots with lines and the outcome resembles Leonardo da Vinci's enigmatic painting of the Mona Lisa.

we looked at two different algorithms – bubble sort and merge sort – and we claimed that merge sort was much more efficient than bubble sort. How can we justify such a statement? This type of question is the business of complexity theory, which examines the computational resources required by an algorithm or class of algorithms. Typically these resources are measured as *time* (the number of computational steps required to solve the problem) or *space* (how much memory does it take to solve the problem). Let us look at the time complexity of our sorting algorithms.

How many operations do we require to sort N objects according to both algorithms? In the bubble sort algorithm, we have to go through the entire list of N objects and perform (N – 1) comparisons. We then have to repeat this process (N – 1) times. To sort a list of length N, we see that for the bubble sort algorithm, the number of comparisons we are required to carry out is:

$$(N - 1) \times (N - 1) = N^2 - 2N + 1$$

Of course there are other statements in the program besides these comparisons, but we are only interested in the behavior of the algorithm for large N. In this case, it is safe for us just to look at the comparisons because the other parts of the program – involving testing and manipulating indices, for example – just take a fixed amount of time. In addition, because for large N, the N^2 term is much larger than the $(-2N + 1)$ term, we can say that the amount of computational work in the bubble sort algorithm applied to N objects grows approximately like N^2. Complexity theorists write this behavior as $O(N^2)$, where the "big-O notation" specifies how the running time of the bubble sort algorithm grows with N.

What about the time complexity behavior of the merge sort algorithm? In this case we used a divide-and-conquer approach and we do not have to cycle through the entire list multiple times. For merge sort, we divide the list up by repeatedly dividing N by 2 and then make comparisons on just the multiple lists containing 2 items. How many times can we divide a list of length N? In our example, we started with 8 items and went from 8 to 4 to 2 so there were three layers and two calls to subroutine Divide. Note that $8 = 2^3$ and we can write the number of layers in terms of logarithms to the base 2. With our more familiar base 10 logarithms, we can write the power of 10 in 1000 as the logarithm $\log_{10} 1000 = 3$. Similarly, we can write the numbers of divisions by 2 for 8 items as the logarithm to base 2, namely $\log_2 8 = 3$ (very handy for binary machines like computers). In general, for N elements we can write the number of divisions as $\log_2 N$.

Because the number of divisions grows like $\log_2 N$ and the number of comparisons we need to make grows like N, the complexity of the merge sort algorithm is $O(N \log_2 N)$. This is the beauty of the divide-and-conquer approach. Table 5.3 shows how the growth rates of N^2 and $N \log_2 N$ compare. We see that $N \log_2 N$ grows much more slowly with N than does N^2 – thus showing the importance of a good sorting algorithm. Any algorithm whose time complexity grows slower than some polynomial – in this case $N \log_2 N$ grows slower than N^2 – is said to be *reasonable*. Any problem for which we can find a low-order polynomial time algorithm is said to be *tractable*, meaning that it can be evaluated by a computer in an acceptable amount of time.

Table 5.3 Growth of operations for sorting algorithms

N	10	50	100	300
N $\log_2$ N	33	282	665	2469
N^2	100	2500	10000	90000

Now let us go back to the traveling salesman problem. We have seen that the brute force method to find the exact solution for the shortest path through N cities grows like N!. A factorial grows with N much faster than any polynomial. As we have seen, we can do better than this brute force solution. Using dynamic programming, in 1962 Held and Karp found an algorithm that solves an N-city TSP in a time proportional to $N^2 2^N$. 2^N corresponds to an exponential time complexity. Exponential growth occurs when the rate of growth of a function is proportional to its current value. As can be seen from Figure 5.17, exponential growth rapidly outstrips linear and quadratic growth, and in fact outstrips any polynomial growth. This means that even though this algorithm to find an exact solution for the traveling salesman problem is much better than our brute force method, it is still unreasonable in that any computational solution will take a time that grows exponentially with N. Any problem for which we can find only exponential time algorithms is said to be *intractable*.

Does P = NP?

Before we leave the subject of algorithmics and complexity, we must introduce one of the most difficult unresolved problems in computer science. It turns out that the traveling salesman problem is representative of a large class of problems that have unreasonable, brute force solutions but for which it cannot be proved whether much faster, reasonable, algorithms exist. These problems are as diverse as devising a timetable to allocate teachers and courses to classrooms with all sorts of constraints; packing items of varying sizes and shapes into fixed-size bins; and determining possible arrangements of patterned tiles. Finding some acceptable solution to even small versions of these problems in real life usually involves much trial and error. After we have made a choice that seemed to be the best possible choice at the time it turns out not to be and we have to backtrack and try some other choice. All of these problems have exponential time solutions, and no one has been able to find an algorithm that solves any of these problems in polynomial time.

The problems in this class are called *NP-complete*. Computer scientists denote the class of all problems that are tractable and have algorithms that take only polynomial time by the symbol P. Besides only having known exponential time solutions, the NP-complete problems have two other important properties: they are nondeterministic, which is what N stands for, and they are complete. To understand what these terms mean let us return to the traveling salesman and pose the problem slightly differently by asking whether or not we can find a tour with a length shorter than a given number of miles. As we have seen, it is very difficult to find the shortest tour, but if we are given a specific tour, it is very easy to verify whether this tour is shorter than the specified length. Where does the nondeterminism come in? Suppose we are trying to find the shortest tour and start out at some city. There are some obvious possibilities for the first step so we toss a coin to decide which city we should visit first. If there are more than two cities to choose from we will have to toss the coin more than once. We now suppose that the coin is not a normal one that just gives a random result, but a "magical" one that always leads to the best choice. The technical term for this magic is *nondeterminism*, and it means that

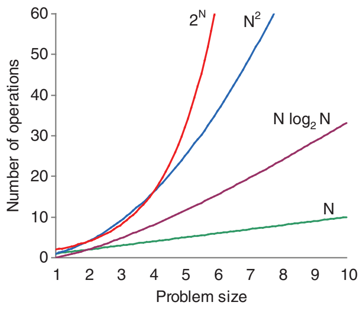
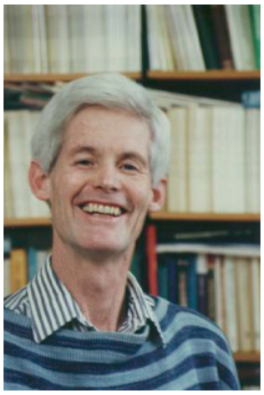


Fig. 5.17. This graph shows the growth with problem size N of four different functions: N; N $\log_2$ N; N^2; and 2^N. The growth of an exponential function like 2^N is much faster than any polynomial like N^2.


B.5.7. Steven Cook received the Turing Award in 1982 for his contribution to algorithmic complexity research.

we do not have to try all the choices to find the right solution. As we can see from the fact that it is easy to check whether or not we have a correct solution, this nondeterministic method can find the solution in polynomial time. This is why these problems are called NP – since there is a nondeterministic polynomial solution.

The second property of NP-complete problems is perhaps the most remarkable. No one has been able to prove that there does not exist a polynomial time algorithm for any of these problems. What the designation *complete* signifies is that if a polynomial time solution were found for one of these problems, then there would be a polynomial time algorithm for all of them! How does this come about? Let us look at another path-finding problem, one that does not involve distances. If we are given a graph consisting of points and edges, can we find a path that passes through all the points exactly once? Such a path is called a *Hamiltonian path*, after the great Irish mathematician William Hamilton. Figure 5.18a shows a Hamiltonian path through five nodes. This problem also turns out to be intractable and NP-complete. Curiously, if we want a path that goes through all the edges exactly once – called an *Eulerian path*, as in Euler's solution to the Bridges of Königsberg problem – the situation is very different. Euler found a polynomial time algorithm for this problem in 1736!

As we have said, the complete in NP-complete signifies that all the problems stand or fall together. Either all NP-complete problems are tractable or none of them are. The concept that is used to establish this is to show that there is a polynomial time algorithm that reduces one NP-complete problem to another. We can see how this works by reducing the Hamiltonian path problem to the traveling salesman problem. In Figure 5.18a we have a graph with five nodes and we have highlighted the Hamiltonian path for this graph. We can construct a traveling salesman network from this graph by using the same nodes, but also drawing additional edges connecting every two nodes as in Figure 5.18b. We assign cost 1 to an edge if it was originally present and cost 2 for each new edge we have added. The new graph has a traveling salesman shortest path of length 6 units – in general N + 1 where N is the number of nodes in the graph – if the original graph had a Hamiltonian path. Thus the answer to whether or not there is a tour no longer than N + 1 is the same as asking whether or not the graph contains a Hamiltonian path. Since the

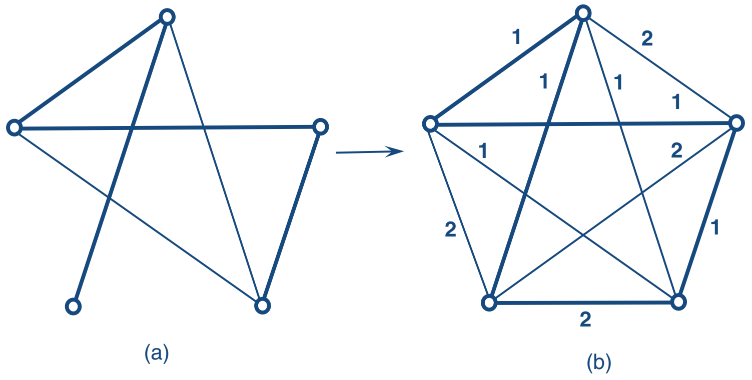


Fig. 5.18. (a) The Hamiltonian path (in bold) connecting five nodes goes through each node exactly once. (b) The Hamiltonian path problem can be converted into a TSP by adding extra edges as described in the text. The traveling salesman tour is shown in bold. (Figure courtesy of David Harel.)

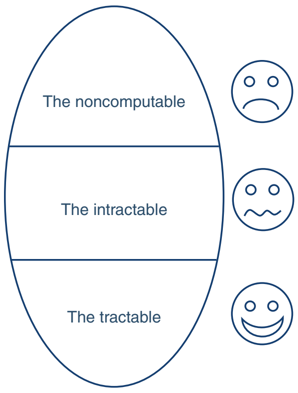


Fig. 5.19. This figure from David Harel's book shows the main problem categories: noncomputable problems have no algorithmic solution. Algorithms for intractable problems do exist but only with exponential or higher order of complexity: tractable problems can be solved with polynomial time algorithms.

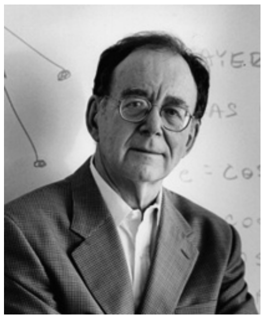
transformation between the two problems takes only a polynomial amount of time, this leads to the claimed result.

We can now see the significance of the title of this section. Figure 5.19 shows how we can divide the world of algorithmic problems into tractable, intractable, and, as we shall see in the next chapter, noncomputable problems. The tractable problems are in the class P and have polynomial time solutions; the intractable problems do not have reasonable polynomial time algorithms. The location of the NP-complete problems is unknown. The question arose from the work of the complexity theorists Steven Cook (B.5.7), Leonid Levin (B.5.8), and Richard Karp (B.5.9) in the early 1970s. Despite more than thirty years of work by computer scientists, the question of whether P = NP is still unresolved.

B.5.8. Leonid Levin discovered the class of NP-complete problems independently from Stephen Cook.

Algorithmics and computability

Numerical simulations of complex physical systems are still a major application area for today's computers. For problems that are very complex, such as weather forecasting or global climate modeling, scientists need to use the fastest, most expensive machines – supercomputers with multiple processors. However, we have also seen how computers can be used to address a variety of different types of problems, from sorting to graph problems. It is here that we have seen the need to use clever algorithms that enable us to solve these problems as quickly as possible. But we have also seen that there are some problems



B.5.9. In 1972 Richard M. Karp wrote a groundbreaking paper, "Reducibility among Combinatorial Problems," in which he identified twenty-one combinatorial problems belonging to the class of NP-complete problems that can be reduced to a common problem – the so-called satisfiability problem. In 1985 he received the Turing Award for his contribution to algorithmic research.

for which no reasonable algorithms exist: the traveling salesman problem is just one of a number of problems for which we know of no polynomial time algorithm. In the next chapter we shall see that there are not only tractable and intractable problems, but also those that are *noncomputable* by any algorithm or computer!

Key concepts

- Algorithms as recipes
  - Euclid's algorithm
- Numerical methods
  - Discrete approximation to continuous variables
  - Monte Carlo method and pseudorandom numbers
- Sorting algorithms
  - Bubble sort
  - Merge sort
- Graph problems
  - Minimal spanning tree
  - Dijkstra's shortest path algorithm
  - Traveling salesman problem
- Complexity theory
  - Big-O notation
  - Polynomial time, tractable problems
  - Exponential time, intractable problems
  - NP-complete problems