Algorithm Analysis and the Transition to Software

Ernst L. Leiss

Department of Computer Science

University of Houston

Houston Texas 77204-3010

elleiss@hotmail.com

713-743-3359 voice

713-743-3335 fax

Key words: Computational complexity, algorithms, performance expectations, efficient software, failures and challenges.

Abstract: Starting with a correct and efficient algorithm, we examine the process of transitioning to software. We outline and illustrate numerous problems that an unaware programmer may encounter. These include incorrect software (producing wrong results), software whose performance differs dramatically from that suggested by algorithm analysis, unpredictable performance of the resulting software, and problems that derive from undecidability and intractability. In many cases, we indicate what a programmer may do to avoid or ameliorate the problems created by these phenomena.
Note: This article is based on the author’s book “A Programmer’s Companion to Algorithm Analysis”; in particular, it uses significant portions of Chapter 4 of this book.

1. Introduction

Our starting point in this article is a correct algorithm for a given problem whose complexity has been thoroughly analyzed. This may be an algorithm from a textbook, a library or collection of algorithms, or a custom-designed algorithm that was developed by the team. We assume that the implementer is reasonably familiar with algorithmic complexity, for the simple reason that without it, one cannot speak of performance in a coherent way. In particular, we assume that both time and space complexities of the algorithm to be implemented are known, usually worst-case, but also average complexities. The time complexity of an algorithm is usually an “operation count” expressed as a function of (some measure of) the input data set (typically its size); it functions as a platform-independent proxy of the wall-clock time the resulting program may require for its execution. Space complexity refers to the amount of memory over and above the input and output data sets that is necessary to execute the program. Average complexity measures the complexity that is expected in an average execution (note however that “average” is a slippery concept and should be carefully defined, preferably in terms of probability distributions of the various cases that may occur, since it may significantly differ depending on the context). Worst-case complexity provides essentially an upper bound that cannot be exceeded.
We assume that the software designer wants to implement this verified algorithm as software. Many programmers have frequently encountered significant problems in this transition. The central focus of this article is an exploration of pitfalls that might occur in the translation of the algorithm into an executable program. We explain the problems and provide explanations and approaches that will either avoid them or ameliorate their effects.

2. Incorrect Software

Often, programmers may start with a correct algorithm, but nevertheless end up with wrong software, in spite of a competent translation of the abstract formulation (the algorithm) into a concrete program (the software). We explicitly assume that the programmer did not make some careless mistake.

The predominant reason is that algorithms specify far less detail than is required for a competent implementation of a program. Below follows a list of contributing factors, which conspire against the unwary programmer.

Exception handling: Algorithms do not concern themselves with minor details, such as division by 0 or the lack of explicit initialization of a variable. In this way, algorithms can concentrate on the essentials and need not worry about being buried in extraneous details. However, software must address adequately all these details; if it does not, incorrect code may result.
**Finite number representation**: At the lowest level, this relates to rounding errors. Most programmers view the issue of rounding errors as one exclusively related to numerical methods. However, there are situations where non-numerical algorithms subtly involve the representation of numbers. A surprisingly difficult issue is testing for equality: In an algorithm, a test whether two values are equal requires no further comment; in a program it may not be as obvious. If the two values are the result of floating point operations in a program, a test for equality may in fact require a more elaborate condition than merely asking whether the two values are identical. For example, the test whether the square of the square root of 2 is equal to 2,

$$\sqrt{2} \cdot \sqrt{2} = 2,$$

fails in many programs (algorithmically, it is of course always true); similarly, the mathematically **divergent** sum $1/2 + 1/3 + 1/4 + 1/5 + 1/6 + 1/7 + \ldots$ **converges** to a fixed constant in virtually all computer platforms.\(^1\) Clearly, in these situations algorithms are based on the mathematical interpretation while programs typically behave according to the rules dictated by the finite representation of the numbers involved and the specific implementation of the arithmetic operations. The gap between these two approaches can result in software that does not produce the results promised by the algorithm; in practical terms, wrong software is obtained. In general, mathematical identities between expressions may not hold in programs, but are of course always assumed to be valid in algorithms. Here is a table that summarizes this, with the reason indicated in parentheses;

---

\(^1\) Even worse, that “constant” number may vary from platform to platform, since it depends on the specific type of arithmetic that a processor implements and the word length of the representation of the numbers (of both the fractions and the sum).
note that $\varepsilon_{\text{max}}$ is that largest floating point number (which is always strictly greater than 0.0) with the property that $1.0 + \varepsilon_{\text{max}} = 1.0$:

<table>
<thead>
<tr>
<th>Identity</th>
<th>Fixed Point</th>
<th>Floating Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Associativity</td>
<td>(a+b)+c = a+(b+c)</td>
<td>Invalid (a, b max, b+c=0)</td>
</tr>
<tr>
<td>(a<em>b)<em>c = a</em>(b</em>c)</td>
<td>Invalid (a, b max, c=0)</td>
<td>Invalid (a, b very large, c very small)</td>
</tr>
<tr>
<td>Distributivity</td>
<td>(a+b)<em>c = a</em>c + b*c</td>
<td>Invalid (a, c max, a+b=0)</td>
</tr>
</tbody>
</table>

Another, even more insidious example is provided by recursion. Consider the following recursive function:

\[
F(x) \\
\quad \text{if } x=0.0 \text{ do } \{ \text{basis case} \} \\
\quad \text{else } \{ \text{statements; } F(x-0.1); \text{statements} \}
\]

As an algorithm, it is quite reasonable, assuming $x$ is a positive integer. However, if it were ever to be implemented as software, it would be an unmitigated disaster. If the function is called with a positive integer as actual parameter (ignoring the mixing of integer and real types), say $F(2.0)$, it will not terminate. This is because the test for equality “$x=0.0$” will fail since 0.1, the decrement, does not have a finite binary floating point representation (even though it does have a finite decimal representation). Thus, the basis case $x=0.0$ will never be encountered and the function will go into an “infinite loop” which will abnormally terminate because the recursion stack will overflow. The standard approach is to replace the test “$x=0.0$” by “$x \leq 0.0$”; this would at least guarantee that the recursion terminates. However, in this case it is virtually impossible to predict what the result will be.
The problem is that we do not know the actual value (that is, the internal, binary representation) of the number 0.1. We know that this value cannot be exactly 0.1, but we don’t know whether it is larger or smaller (admittedly by very little, but this very small quantity nevertheless trips us up). The obvious intent of the original algorithm is to invoke recursion 10 times for each integer unit; thus F(2.0) should give rise to 20 recursive calls, namely F(1.9), F(1.8),..., F(0.2), F(0.1), and finally F(0.0) which then invokes the basis case. However, if the actually represented value of 0.1 is smaller than the real value 0.1, then the actually represented values of the intermediate quantities (1.9, 1.8, ..., 0.2, 0.1, and 0.0) will all be larger than the values of their “identifiers”.2 As a result, it will not be this last value that invokes the basis case (since it is objectively larger than 0.0), but the next value computed in this sequence which is something quite close to –0.1. Conversely, if the actual, binary value of 0.1 is larger than 0.1, it will have the expected number of recursive calls. We do not know what the internal representation of 0.1 is; therefore the result of the program is unpredictable. It gets even weirder: Since programs are typically not cognizant of the bit length of the processor, we have the expectation that in general, a 32-bit processor will produce substantially similar results as a 64-bit processor. However, in this case it is conceivable that the 32-bit representation of 0.1 is smaller than 0.1, but the 64-bit representation of 0.1 is larger than 0.1 (or vice versa), resulting in differing number of recursive calls, depending on the word length!

A related notion is numerical stability: some algorithms solve certain numerical problems but are unsuitable to be implemented in code. Almost invariably this is due to the finite

---

2 This is not absolutely guaranteed, although extremely likely: We are assuming monotonicity here, that is subtracting something smaller than 0.1 from 2 yields something larger than 1.9, and then subtracting something smaller than 0.1 from this value yields something larger than 1.8, etc..
length of the numbers involved in the computations. This is a topic that is extensively studied in numerical analysis and is well known there.

**Passing parameters:** Algorithms typically do not specify all details of the interface between the calling program and the algorithm itself. This implies in particular that the question of how to pass the arguments is frequently not adequately addressed in the abstract (algorithmic) formulation. However, in the transition from algorithm to software, the algorithm must be encapsulated as a function, and this function must receive its arguments as parameters. The communication of these arguments from the calling program to the function is known as parameter passing. There are several, fundamentally different ways of passing parameters, and it is crucial to understand them, since using an inadequate method of passing parameters to a function may result in code that produces wrong results. Here is an example.

Many programming languages offer two fundamentally different ways in which parameters are passed, call by value and call by reference. (There is a third one, call by name, which is ignored here since it is rarely used.) The programming language defines how parameters are passed, specifically the way in which the actual parameters of a function call are tied to the formal parameters.

**Call by value** assumes that the actual parameter has a value and that the formal parameter corresponds to a (physical) memory location. The actual parameter is evaluated and the resulting value is copied into the memory location corresponding to the
formal parameter. The execution of a successful function call proceeds then as if the formal parameters were local variables of the function, except that they had been assigned initial values before the instructions in the body of the function are executed. As is true for all local variables, the formal parameters of a function are no longer accessible once execution of the body of the function has ended and control is returned to the calling program.³ Note that the space required for the formal parameters increases the space complexity of the resulting program. Similarly, the copying (of the values of the actual parameters to the locations of the formal parameters) increases the time complexity.

Call by reference assumes that each actual parameter corresponds to a memory location. The corresponding formal parameter is then associated with this memory location for the duration of execution of the function call. Thus, the use of a formal parameter in an instruction in the body of the function results in the use of the memory location that was assigned to the corresponding actual parameter. Since the memory space allocated to the actual parameters is used also by the formal parameters, call by reference tends to require less space.

Call by value is considered a safer way of passing parameters than call by reference because passing parameters by value cannot affect anything in the calling program, except for the result that is reported back from the function. In contrast, call by reference can modify values in the calling program, since memory locations of the calling program (those of the actual parameters) are manipulated by the function.

³ Any space no longer accessible can be returned to the pool of available dynamic memory. This means in particular that when invoking a function twice, there is no guarantee that the same space is allocated again.
When designing an algorithm for a specific problem, one ignores the details how this algorithm is tied into the overall program since the interface between the resulting algorithm, now encapsulated as a function, and the algorithm from which this function is called is usually of little concern. This changes as soon as the algorithm is implemented as a program in a specific programming language.

Consider the standard matrix multiplication algorithm, consisting of three nested loops, which assigns the product of the two \([1:\text{n},1:\text{n}]\)-matrices \(A\) and \(B\) to the matrix \(C\) of the same type:

\[
\text{for } i:=1 \text{ to } n \text{ do }
\quad \text{for } j:=1 \text{ to } n \text{ do }
\quad \{ \text{\quad } C[i,j]:=0; \text{\quad for } k:=1 \text{ to } n \text{ do }
\; \quad \text{\quad } C[i,j] := C[i,j] + A[i,k]*B[k,j] \}
\]

When encapsulating this algorithm into a function, say, \(\text{MatMult}\), the three matrices \(A\), \(B\), and \(C\) (as well as \(n\)) must obviously be parameters:

\[
\text{function MatMult}(A,B,C,n) 
\quad \{ \text{\quad for } i:=1 \text{ to } n \text{ do }
\quad \quad \text{for } j:=1 \text{ to } n \text{ do }
\quad \quad \{ \text{\quad \quad } C[i,j]:=0; \text{\quad \quad for } k:=1 \text{ to } n \text{ do }
\quad \quad \quad \text{\quad \quad } C[i,j] := C[i,j] + A[i,k]*B[k,j] \}\}
\]

How are these four parameters to be passed? The variable \(n\) can be passed by value since only its value is relevant. For the three matrices, call by value does not work, since it does not allow the reporting back of the result (the matrix \(C\)) to the calling program – recall that upon completed execution of the body, the local variables (including the formal
parameters) are no longer accessible. Moreover, copying a $[1:n,1:n]$ matrix requires time proportional to $n^2$. Thus, we may conclude that call by reference is appropriate for the three matrices. This turns out to be wrong! Consider the following entirely legitimate function call (essentially squaring the matrix $X$ and assigning the result again to $X$), where $X$ is a $[1:n,1:n]$ matrix:

$$\text{function MatMult}(X,X,X,n)$$

If we pass our three array parameters by reference, our code is wrong! To see this consider the following matrix $X$:

$$X = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \text{ the square of } X (X \times X) \text{ is } \begin{pmatrix} 7 & 10 \\ 15 & 22 \end{pmatrix}, \text{ but } \text{MatMult}(X,X,X,n) \text{ yields } \begin{pmatrix} 6 & 0 \\ 0 & 0 \end{pmatrix}.$$

One can show that the matrices $A$ and $B$ must be passed by value and the matrix $C$ by reference for this code to be correct. This incurs of course an additional space complexity of $2n^2$ since space for the two (local) matrices for $A$ and $B$ must be provided and an additional time complexity (beyond the $O(n^3)$ that the three nested loops require) of the same amount, $2n^2$, since the actual parameters must be copied into these memory locations. By modifying the code in a small way, one can get a more efficient solution that requires only $n^2$ additional space and only $n^2$ additional time.

**Implicit assumptions:** When designing an algorithm we may make reasonable assumptions, which nevertheless may be violated by the resulting program under some circumstances. For example, we may assume that a sorting algorithm does not change the

---

4 Most programming languages do not allow the result of a function to be anything but an object of a simple type. Thus in particular, the result of our function MatMult cannot be an array.
order of the elements of an array if that array was already sorted. If a change in the original position of identical elements is undesirable, this would have to be stated explicitly, but this requirement may have been overlooked.

While it is somewhat unusual for the code derived from a correct algorithm to fail to produce the expected results, it occurs more often than desirable. Note that we deal here with sequential algorithms; parallel algorithms are much more likely to give rise to incorrect parallel programs.

3. Parallelism

Parallelism is an aspect of software with which programmers are generally unfamiliar. However, virtually all modern computing systems exhibit hardware parallelism. While writing parallel programs is probably not imminent for most application programmers, it is nevertheless useful to have some knowledge of the underlying software principles.

Parallel architectures are used because of their promise in increased performance. At the most primitive level, if two or more devices operate at the same time one would expect to improve the overall performance of the system. There is a wide spectrum of different

---

5 Sorting methods that guarantee that they will not interchange the order of identical elements are called stable. The fact that a name was coined to differentiate them from those that might swap such elements indicates that this aspect is quite important.

6 One of the most insidious problems of parallel (and distributed) software are so-called race conditions whereby two processes compete for some resource (e. g., access to memory, communication links, I/O controllers) – and sometimes one wins, at other times the other wins, even though the starting configurations of the two instances are seemingly identical.
models of parallelism, from vector computing to shared-memory MIMD systems, to distributed memory MIMD systems. Each requires specialized knowledge to allow programmers to exploit them efficiently. Common to most is the quest for speed-up, a measure of the improvement in time obtained by using several hardware devices in place of a single one.

In general, speed-up is both dependent on the specific architecture and on the quality of the algorithm. Different architectures may permit rather differing speed-ups, independent of the quality of the algorithm. In fact, it may be virtually impossible to take an algorithm that works well on a particular parallel system and apply it effectively to a different parallel architecture.

Parallel algorithms frequently assume the shared memory paradigm, that is, they assume that there are several processors but only one large memory space, which is shared by all processors. From a theoretical point of view, one can differentiate two types of access to a unit of memory, exclusive and concurrent. Exclusive access means that only one processor may access a specific memory unit at a time; concurrent access means that more than one processor can access the memory unit. Further, two types of access can be distinguished, reading and writing. Therefore, we can image four types of combinations:

\[ \text{EREW, ERCW, CREW, CRCW} \]

where E stands for exclusive, C for concurrent, R for read, and W for write. Of these four, the first, EREW, is the standard mechanism implemented in all commercial systems.

---

[7] Michael Flynn defined a classification of parallelism by concentrating on instruction streams (I) and data streams (D); the presence of a single stream (of type I or D) is then indicated by S, that of multiple streams by M. This gives rise to SISD, SIMD, MISD, and MIMD systems.
(including all parallel shared memory systems). The second, ERCW, makes very little sense, since it is writing that is difficult to image being carried out in parallel. However, the third, CREW, is conceptually quite sensible – it simply means that several processors can read a unit of memory at the same time.\(^8\) However sensible concurrent reading may be, no commercially successful computing system has implemented it; therefore it is of no practical significance. Theoretically, CRCW is most powerful, and CREW is less powerful than CRCW but more powerful than EREW. However, these results are only of a theoretical nature and have no practical significance at all.

An alternative to the shared-memory approach assumes that each processor has its own (private) memory and communication between processors relies on message passing. In this situation, it is necessary to specify what messages are sent and at what time. While this creates significant problems for the programmer, it does not provide new programming paradigms that must be considered. Therefore, it does not give rise to new complexity considerations.

It should not come as a great surprise that programming parallel systems is significantly more difficult than programming ordinary, sequential systems. However, when designing algorithms (or producing code), one must distinguish between covert and overt parallelism. In covert parallelism, the designer ignores the parallel nature of the hardware and designs a standard sequential algorithm; it is only for overt parallelism that parallel algorithms must be devised. Here we are concentrating on sequential algorithms; they are

---

\(^8\) This occurs very frequently in practice, in different contexts: Consider a movie theater where many patrons see (that is, read!) the same content at the same time. Clearly, writing is a completely different issue!
not parallel, even though the hardware on which the software ultimately executes may well contain a great deal of parallelism. Any exploitation of the available parallelisms in the hardware would be done by the compiling system, the operating system, or the run-time support system, all of which are typically outside of the program designer’s influence.

What is the promise of parallel hardware? Recall the notion of speed-up. If we have \( p \) processors instead of one we might hope for a speed-up of \( p \) – after all, there is \( p \) times more hardware available! This ignores the threefold bane of programming parallel systems: Overhead, lack of balance, and synchronization. As a result, the ideal speed-up, of \( p \) for \( p \) processors compared with 1 processor, is almost never attained. In many cases, significantly lower ratios (for MIMD systems perhaps 50\% for smaller \( p \), say \( p \leq 32 \), and 20\% or less for \( p \) on the order of thousands) are considered very respectable. An additional complication stems from the fact that a good parallel algorithm is not necessarily obtained by parallelizing a good sequential one; there are cases known where parallelizing a *bad* sequential algorithm produces a much better parallel one!

### 4. Performance Discrepancies

When one designs an algorithm, one typically also determines its time and space complexity. These measurements provide a useful indication how long an execution with a given data set should take (time complexity) and how much memory an implementation will require (space complexity). Discrepancies between the performance obtained from
the analysis of the algorithm and the observed performance of the code obtained by translating the algorithm faithfully are more common than outright wrong results. However, the reasons are sometimes quite similar. For example, choosing an inappropriate way of passing parameters may seriously affect the performance of the resulting program. Binary search provides a particularly egregious instance since the wrong parameter passing mechanism can slow its performance exponentially! Specifically, binary search on an array with \( n \) elements runs in time \( O(\log_2(n)) \), but if one were to pass that array by value, the copying process involved is of time \( O(n) \), which is exponentially slower than the actual search. However, much more common causes are the memory hierarchy of modern computing architectures and the support systems (compilers, operating systems, run-time execution systems) that heavily influence how efficiently a program executes. Most importantly and in contrast to the binary search example, problems arising from the memory hierarchy are completely hidden from the programmer.

The memory hierarchy is a major culprit implicated in performance discrepancies. Algorithms assume a flat main memory space that is uniform and unlimited; programs of course exist in a universe where memory is limited and comes in a variety of forms, from very fast (registers) to very slow (external memory). Typically, the faster the memory, the more expensive it is. Many programs tend to use virtual memory management (VMM) to give the programmer the illusion of having more memory available than is really there. This is achieved by using external memory which however is significantly slower.\(^9\) The

\(^9\) If magnetic disks are used for external memory, the difference can be seven orders of magnitude. Even if solid-state devices are used in place of magnetic disks, they are still (at least) one order of magnitude
major problem is that VMM interacts fairly subtly with other aspects of a program. Consider the seemingly trivial problem of adding two matrices:

\[ C := A + B \]

where A, B, and C are matrices of type \([1:n,1:n]\). If \(n\) has a value small enough that the three matrices fit into the available memory, this is called an in-core problem. In-core problems are very predictable since they do not involve the memory hierarchy. However, in-core problems, also often known as toy problems, are usually not very useful. Much more common in real life are out-of-core problems; in our example this means that \(n\) is large enough that the matrix addition cannot be carried out in main memory. For the algorithm, the above formulation is fully sufficient; to understand the program execution, we need to know a good deal more. A typical program fragment might look like this:

```
for i:=1 to n do
    for j:=1 to n do
```

Since main memory is one-dimensional, the two-dimensional arrays A, B, and C must be mapped into main memory. (This is true for both in-core and out-of-core programs.) Programming languages (or more specifically, their compilers) provide two standard mapping functions for this purpose, row-major and column-major. If we don’t have enough main memory to accommodate the three matrices, the mapping function will map each array into the logical memory space, which in turn is divided up into blocks, and it is these blocks (called pages) that are fetched from external memory and stored to slower than main memory. The trend in high-performance systems is to prohibit the use of disks for scratch space during execution (that is, only input data sets may be read from disk and output data sets may be written to disk, but no intermediate memory requests may employ magnetic disks); this is dictated by the slowness of magnetic disks drives whose access speeds have essentially not changed over the past two decades, the reason being their mechanical nature. In contrast, electronic devices such as processors, caches, and main memory get faster almost every year.
external memory by VMM. Assume that $n=2^{13}$ (8K), that the size of a page is $2^{11}$ (2K) words, and that our available main memory permits us to have $2^{10}$ (1K) pages in main memory. If the memory mapping function is row-major, each row consists of four pages; if it is column-major, each column consists of four pages. Since the total amount of space required for the three matrices is about $3\cdot 2^{26}$ (or about 200M) words but only $2^{10}$ pages are available, VMM will swap pages in and out of main memory as dictated by the code above.

Here arises the first problem: Most programmers do not know which memory mapping function is used.\footnote{A rule of thumb is the following: A language directly based on Fortran uses column-major, all other languages use row-major memory mapping. However, it is a good idea to make sure of this – some programming languages do not specify explicitly which mapping function is to be used by the compiler, and so this becomes a compiler property (with obvious implications for portability).} Therefore, they are unable to determine how many pages this very simple program fragment will swap in and out. In fact, most programmers are not particularly keen on understanding VMM.\footnote{Indeed, for many programmers, the most important aspect of VMM is precisely that it permits them to ignore input/output problems!} For our explanations, we assume that the replacement policy is pure LRU (Least Recently Used); this means that the page that has been unused for the longest time will be the next to be swapped out if the need arises to bring in a page but all available memory space is occupied. Most common operating systems that support VMM implement some version of LRU.

There is another, even more serious problem: Most programmers believe all this information is of no relevance to writing good code. They would be correct if the three matrices fit into main memory (in-core). However, if they don’t fit into the available
main memory, the difference between the numbers of pages swapped for one and for the other mapping function is staggering. Specifically, if the memory mapping function is row-major, $3 \cdot 2^{15}$ (about 100K) pages are swapped in and out, but if it is column-major, it is $3 \cdot 2^{26}$ (about 200M) pages! In other words, one version swaps fewer than 100,000 pages, the other almost 200 million! Thus, it is safe to assume that one version is about 2,000 times slower than the other. Or, to be even more drastic, if the faster version takes 15 minutes to execute\textsuperscript{12}, the slower might take about three weeks if magnetic disks are involved. Yet, from an algorithmic point of view, the two versions have identical performance since the algorithm assumes uniform memory access. Of course, as soon as external memory, or more specifically VMM, comes into play, the uniformity vanishes.

Of course in practice, no sane programmer would let a program execute for weeks, if the running time predicted on the basis of the algorithm was dramatically less.\textsuperscript{13} Instead, the programmer would assume after a while that “something went wrong” and abort execution. Thus, it would appear that the code is “wrong”, merely on the basis of the performance expectations that were derived from the analysis of the underlying algorithm.

\textsuperscript{12} This may actually be pushing it – if the external memory is a magnetic disk drive, it takes typically about 10 ms to retrieve a page, so in 15 minutes fewer than 90,000 pages can be retrieved, assuming that nothing else happens. In the future, we will probably dispense with magnetic disk drives for external memory, which will reduce the overall timings, but the number of pages to be retrieved is independent of the type of external memory employed. Since the replacement of magnetic disks will still be slower than main memory, the observations still hold. So, the over-all wall-clock time may not be weeks, but it will still be numerous hours (as opposed to 15 min) assuming the external memory is ten time slower than the main memory.

\textsuperscript{13} Indeed, time complexity which is based exclusively on operation counts indicates that there are $n^2$ statements to be executed. If one assumes that each addition and each assignment takes 10 ns, then this would suggest a running time of 2 seconds, which is extremely misleading because it ignores the data movements required by the code!
There are techniques that permit a programmer to avoid such problems. In fact, it is fairly easy to change the code in such a way that the worst case behavior above (out-of-core; A and B must be fetched from disk and C must be written to disk; the memory mapping function is column-major) can be avoided. More intriguingly yet, using optimization techniques as they are available to any good optimizing compiler, such code changes can even be done automatically! In other words, if the memory access that the program prescribes clashes with the memory mapping function, a good optimizing compiler could automatically carry out a program transformation that results in much better performance. Such program transformations must be semantically valid, that is, they do not change the meaning of the resulting program. This is based on the theory of dependence analysis which determines which program transformations (of a given catalog) are semantically valid. To be specific, for the given sample code,

\[
\text{for } i := 1 \text{ to } n \text{ do }
\text{for } j := 1 \text{ to } n \text{ do }
\text{C}[i,j] := \text{A}[i,j] + \text{B}[i,j]
\]

if the memory mapping is row-major, it will be efficient, but if the memory mapping is column-major, it will be extremely inefficient. Thus, if column-major is used (this is something not under the control of the programmer as it usually is defined by the programming language used), one can transform this code fragment into this semantically equivalent code fragment:

\[
\text{for } j := 1 \text{ to } n \text{ do }
\text{for } i := 1 \text{ to } n \text{ do }
\text{C}[i,j] := \text{A}[i,j] + \text{B}[i,j]
\]

which will be as efficient as the previous one for row-major. This transformation is called a loop-interchange because the i-loop and the j-loop are interchanged, and it is
semantically valid because there are no dependences among the statements executed in the loops.

5. Unpredictability

As noted, the complexity of an algorithm provides a good predictor for the resource demands of a program based on that algorithm. It is therefore disturbing when the execution of the resulting software does not conform to the predictions obtained by analyzing the underlying algorithm. This is particularly annoying if the observed performance of the software changes seemingly randomly from one run to the next, even though the input data are similar (or even identical).

Here is an illustration. Assume we have developed an algorithm to solve a given problem. This algorithm has a certain time complexity $f(n)$ where $n$ is the size of the input. This function $f(n)$ is typically obtained by determining the operation count of the algorithm. Now assume that translating this algorithm into code results in a program that is being executed repeatedly, with very similar data sets. When measuring the run time of the program we may find that for identically sized input sets, very different timings of the program executions are observed. Moreover, nothing in the time complexity analysis of the algorithm indicated that the range of $f(n)$ for a given value of $n$ should be more than minimal. How can this discrepancy between time complexity and run time behavior be explained?
Clearly, the issues of non-uniform memory access outlined in the previous section could be in play. However, if the amount of main memory available exceeds the space complexity of the algorithm and if the program is a reasonably faithful implementation of the algorithm, such issues do not explain how different runs of a program with identical memory requirements could have differing execution times. Instead, the culprit is often the management of dynamic data structures.

Many algorithms assume the availability of intermediate data structures that are used in the process of producing the desired result, but have no further significance. For example, in MergeSort, we need an additional array to store intermediate results (consisting of sequences of sorted subarrays which are then to be merged to create larger sorted subarrays). This additional array is required by MergeSort during the process of sorting but can be discarded once sorting is completed. Similarly, when deleting an element from a binary search tree, the action of discarding a node is carried out by disconnecting the node from the tree, thereby making that node inaccessible. In both cases, at some point memory is freed up; this means in particular that this memory can be used subsequently for other purposes. Thus, it should be clear that the space complexity of sorting m arrays of size n using MergeSort is O(n) and not O(m*n) – the intermediate array needed for the first sort can be reused in the next sort, and so on. Similarly, a search tree with n nodes may be the result of many different insertions and deletions, say n_{ins} insertions and n_{del} deletions with n=n_{ins}-n_{del}; however, the space complexity of representing this search tree is always assumed to be O(n), and not O(n_{ins})! Therefore, the complexity analysis of the algorithm implicitly assumes an idealized world in which space that is no longer needed
is automatically thrown back into a pool of available space, without anybody having to attend to this action.\textsuperscript{14} We want to note here as well that recursive functions require additional memory to implement recursion, namely the recursion stack: Clearly upon returning the space used for the recursion stack is released but in algorithms it is typically not reused for other data structures.

When transitioning from algorithm to software, an explicit mechanism must exist that allows the reuse of memory once it is freed. This is necessary because in contrast to the idealized algorithm world where memory is unlimited and therefore almost free, in the real world of software, memory is limited and must not be wasted gratuitously. The invocation of such a mechanism may have to be done explicitly by the programmer or it may be automatical. Some programming languages require the programmer to attend explicitly to the freeing (deallocating, releasing) of memory no longer needed by the program and provide statements for this purpose; these statements for freeing memory are viewed as paired up with the statements that earlier in the program execution allocated the memory for the dynamic data structures. Thus, it is expected that the programmer provide statements that free no longer needed space. Other programming languages assume that the collection of freed-up space is carried out automatically, by a process known as garbage collection. Finally, memory used by the recursion stack is also freed up once the recursive function returns.

\textsuperscript{14} Note that, in contrast to many programming languages, algorithms do not stipulate that discarded memory be explicitly put back for later reuse. Since memory in the algorithm paradigm is unlimited, such an action would be unnecessary and perhaps even confusing.
In both cases, unpredictability may result. Consider first the situation where the programmer must free memory explicitly. It is of course true that at the end of executing a program, all space allocated to that program is freed. This may suggest to a programmer that the tedious business of explicitly freeing memory during execution is not necessary; this would be very wrong since the program could very quickly run out of available memory and crash. Alternatively, an inexperienced programmer may not even be aware of the need to write statements to free memory. If this occurs in a programming environment where a fixed amount of memory space is allotted to a program, then exceeding that space would again crash the program. However, this is less likely to happen, since many computer systems provide virtual memory management. Thus, if the amount of freed space is not excessive, it is quite possible that VMM avoids crashing the program. But as we pointed out above, VMM can seriously slow down the execution of a program, with the outcome that seemingly highly similar input sets result in vastly different paging requirements for VMM. Consequently, the timings can differ greatly as well.

Consider now the case where garbage collection is carried out automatically by the runtime support system. In this scenario, it is the garbage collection system that decides autonomously when to do garbage collection. It is not very difficult to see how unpredictability can arise – the programmer simply does not know when garbage collection is carried out. Note that the process can be quite time consuming (it may involve fairly complex algorithms, depending on the nature of the data structures involved in freeing memory); moreover, it may interact with other processes running on
the same system which have nothing to do with our program. Thus, even if our program executions are identical in instructions and input, the system configuration may differ from one run to the next, implying that garbage collection may not occur at the same times or with the same frequency. The most insidious aspect of garbage collection is the following: The complexity of garbage collection is proportional to the total size of the main memory available. Thus, the standard argument of naïve programmers, namely that one should just get more memory to avoid problems with garbage collection, backfires badly: Yes, it is indeed true that with larger amounts of memory, garbage collection will occur less frequently, but when it occurs (and it will occur!), it takes longer, thereby rendering the program behavior even more unpredictable!

Finally, we mention that dynamic memory allocation can happen even if the programmer thinks that no dynamic structures are created in the program. The problem is that every call to a recursive function involves a dynamic recursion stack. Thus, every program containing recursive function calls necessarily must deal with freeing memory. Since the actual deallocation of this memory is outside of the programmer’s influence, freeing recursion stack memory must be done automatically. Since the size of the recursion stack is often unpredictable (consider search trees, for example, where we can bound from above the size of the stack, but the actual size depends on the specific tree), whether or not garbage collection in this context is carried out, and how often, may well be quite unpredictable. While it is not possible to avoid the problems that arise out of unpredictability, programmers should at least be aware of the implications and be prepared for them.
6. Infeasibility and Impossibility

Our last topic is fundamentally different from the previous ones, but usually a disappointment nevertheless. This is the case when the theory of algorithms tells us that a certain problem does not have a solution; the problem is unsolvable or undecidable. Alternatively, there may be a solution, but its time complexity is so large that it is unrealistic to expect a solution; the problem is intractable. On the other hand, the programmer may feel quite strongly that the problem she wishes to solve is a reasonable one, one that arose in the context of an eminently sensible problem setting. We contend that in some cases the difficulty is not the problem itself, but a seemingly sensible generalization of the problem at hand. Reverting to the original problem or restricting it to the cases one is really interested in frequently allows one to circumvent the problems of undecidability or infeasibility.

A typical example is provided by compilers. Another example of an undecidable problem is provided by the question whether a given program contains an infinite loop.
phase of the compiler) than others. Therefore, it is quite sensible to change the original context free grammar into one more amenable for the purpose of parsing it; needless to say this new grammar must generate the exact same programming language. Ordinarily, when going from one grammar to another, certain rules are applied which have the property that the resulting grammar is equivalent to the original one; however, one might wish to be able to verify independently that the two grammars do indeed generate the same language. Thus, instead of verifying the equivalence of transformation rules, one might come up with the idea of writing a program that accepts as input the two context free grammars and determines whether they are equivalent. Since it appears to be easier to solve this problem in general, the designer of the program might decide to refrain from imposing restrictions on the two grammars. This is where things go badly wrong: It is known that such a program cannot exist. More specifically, the problem of determining whether two arbitrary context free grammars are equivalent is undecidable. Undecidability means that one can prove with mathematical rigor that no algorithm for the problem at hand can exist, now or ever. This is quite surprising in the context of our grammar problem, but the root cause is the undue generalization of the problem. Had we restricted our attention to grammars that were “similar” to the original (that is, were obtained in some transformational way from it), the problem would most likely have been solvable. It is only the decision to consider two arbitrary context free grammars as input that rendered the problem undecidable. In other words, it is often possible to define interesting subcases of the given problem which do have solutions.
There is another aspect of impossibility, namely extremely large time complexity. While this is not as dire as undecidability (which says we cannot solve the general problem under any circumstances), from a practical point of view, it can be equally serious. Algorithms with a time complexity $O(n^c)$, for $c$ some fixed constant and $n$ the size of the input set, are said to have polynomial time complexity. Clearly, algorithms with complexities that are not polynomial are practical only for relatively small values of $n$; they are infeasible if $n$ is large. So what is a programmer to do when faced with the problem of implementing an algorithm with a prohibitively high time complexity?

In many cases, the programmer focuses on the worst case time complexity. While the worst case complexity is an appropriate measure if one is interested in an upper bound, in practical situations the average complexity may be much more sensible (unless we are dealing with a hard real-time applications, such as the control of a nuclear power reactor or air-traffic control). Some algorithms have rather substantial gaps between the worst case and the average complexity.

Probably the most celebrated algorithm with an astonishingly large gap between the worst case and the average time complexity is the Simplex algorithm for linear programming. This algorithm is known to have an extremely bad worst case time complexity.

---

16 Obviously, extremely large space complexity would also be a concern, but it makes little sense to have an algorithm in which the time complexity is smaller than the space complexity. Therefore an algorithm with an extremely large space complexity most likely has also an even larger time complexity.

17 In linear programming, we are to minimize (or maximize) a quantity subject to certain constraints where both the quantity and the constraints must be linear functions of the $n$ variables $x_1, \ldots, x_n$. For given constants $a_{ij}$, $b_i$, and $c_i$, minimize the linear function $c_1x_1 + \ldots + c_nx_n$ subject to the linear constraints $a_{ij}x_1 + \ldots + a_{in}x_n \leq b_i$ for all $i=1, \ldots, m$. Many practically important problems can be formulated as linear programming problems. As a result, there is a vast body of knowledge as well as algorithms focusing on these problems.
complexity: there exist input sets for which the algorithm requires \textit{exponential} time! Yet, for almost all practical problems it runs in \textit{linear} time. Most interestingly, there do exist other algorithms for solving linear programming problems that have a polynomial time complexity, but their average time complexity tends to be slower than that of the simplex method. Therefore the Simplex method is still the most widely used algorithm for solving linear programming problems.

The upshot of this discussion is that the worst case time complexity may be the wrong way of looking at an algorithm when implementing it as software. Instead, it may be far more reasonable to determine what are the practically significant cases to which one applies the program and then to determine what the average time complexity is for these cases.

Finally we observe that we may ask for something that we are not really that interested in. Specifically, we are conditioned to concentrate on optimal solutions and often ignore that optimality may not be inherently needed in a particular context. Specifically, consider a case where finding an optimal solution is infeasible but determining an approximation is not. It should be clear that there are instances where optimality it not the driving force; often, people are content if they can obtain a solution that is close to the optimal one, that is, a “good enough” solution. However, not every problem where it makes sense to talk about an approximate solution admits approximations that are acceptably good. Here is an illustration.
The bin packing problem can be stated as follows: We are given a set of n rational numbers $w_1, \ldots, w_n$ such that for all $i=1, \ldots, n$, $0<w_i<1$; the problem consists of packing the numbers into the minimum number of bins such that the sum of all numbers in each bin does not exceed 1.\(^{18}\) The problem of determining the minimum number of bins required is known to be infeasible to solve.\(^{19}\) However in most practical situations, a sufficiently good approximation is probably acceptable. This obviously gives rise to the question what is “sufficiently good”? It turns out that in the problem at hand, bin packing, we have an excellent answer: we can guarantee that the approximate, but very efficient solution is within a certain fixed percentage of the optimal one which takes essentially exponential time (in $n$). This penalty in the case of bin packing is 22%; in other words, we have a very efficient algorithm (it requires time $O(n^2)$) and provides us with a solution that is never more than 22% worse than the optimal solution which requires time exponential in $n$.

Certain classes of algorithms permit approximate solutions. In all such cases, heuristics play an important role. Some of these algorithms provide provably good solutions – by this we mean that their solutions are within a constant factor of the optimal one. Other problems may have approximation algorithms but they do not have provably good solutions; rather their solutions can differ from the optimal solution by an arbitrary large constant factor. In most cases, such algorithms are less useful. However, their

\(^{18}\) This is the simplest, one-dimensional formulation. Bin packing has many generalizations, for example to more than one dimension.

\(^{19}\) Specifically, it is NP-complete, meaning that for practical purposes, any algorithm solving it optimally has a time complexity that is exponential in $n$. 

29
deficiencies should always be viewed in the light of the infeasibility of the algorithms providing exact, optimal solutions!

7. Conclusion

We outlined and illustrated the more important sources of disappointments that one may encounter when translating an algorithm into software. Even though we assume explicitly that the starting point of our discussion, the algorithm, is correct, it is still possible to end up with an incorrect program. We then zeroed in on performance, in particular factors that can cause a major discrepancy between the complexity that the analysis of the algorithm supplied and the observed performance of the resulting program. Related to this is the issue of unpredictability, specifically of performance, whereby seemingly identical program executions may display rather substantial performance differences. We also commented on the role of parallelism in software development. Finally we addressed issues related to impossibility, either provable impossibility, also known as undecidability, or practical impossibility, namely where the time complexity is so large to be essentially prohibitive for practical consumption.

Bibliographical Notes

Collecting references for the material on which this chapter is based provides an excellent indication why software practitioners find it difficult to understand how algorithms fail to translate properly into software: There are many rather diverse topics that come to bear
on this issue. Moreover, these topics tend to be taught in different courses in a college setting and are covered by different textbooks, making it even more difficult for the interested software designer to understand all the ramifications for effective programming. Apart from the obvious, namely diverse programming languages and algorithm analysis, the areas implicated are numerical analysis, programming language design, compiler design, and operating systems. While it is not necessary to be on the forefront of research in each of these areas, it is important to have a reasonable understanding and working knowledge of all of them.

Exception handling is covered by the language manual of the programming language employed. Rounding errors, stability, and other aspects of numerical errors analysis and propagation are comprehensively covered in standard textbooks on numerical analysis and methods, for example Higham: *Accuracy and Stability of Numerical Algorithms*; see also the paper by Goldberg, entitled What Every Computer Scientist Should Know about Floating-Point Arithmetic. Methods for passing parameters as well as memory mappings are traditionally covered in three quite different places, in the language manual of the programming language employed (but this tends to be cursory and only focused on that language), in comparative programming language books, and, perhaps most detailed, in standard compiler text books (for example Aho, Sethi, and Ullman: *Compilers: Principles, Techniques and Tools*), since it is the compiler that has to grapple with the actual implementation of the various mechanisms stipulated by a programming language. Virtual memory management on the other hand is part of the coverage of operating systems; we refer to standard textbooks on this topic, for example Silberschatz, Gavin,
and Gagne: *Operating Systems Concepts*. The same applies to garbage collection which is usually part of operating or runtime support systems. Undecidability and intractability on the other hand are part of the general algorithm repertoire that is covered in textbooks on the design and analysis of algorithms, for example Kleinberg and Tardos: *Algorithm Design*.

This paper is based on E. L. Leiss: *A Programmer’s Companion to Algorithm Analysis*; the only other books treating similar material are the two *Programming Pearls* collections of papers by Jon Bentley, referenced below.


