PRIFYSGOL glyndŵr UNIVERSITY

Glyndŵr University
Glyndŵr University Research Online

Computing | Computer Science

11-1-2006

Complexity Issues in Control Software Design: A practical perspective

Stuart Cunningham
Glyndwr University, s.cunningham@glyndwr.ac.uk

Vic Grout
Glyndwr University, v.grout@glyndwr.ac.uk

Follow this and additional works at: http://epubs.glyndwr.ac.uk/cair

Part of the Computer and Systems Architecture Commons, Digital Communications and Networking Commons, Hardware Systems Commons, and the Systems and Communications Commons

Recommended Citation
Cunningham, S. & Grout, V., (2006), 'Complexity Issues in Control Software Design: A practical perspective'. [Paper presented to the 19th ISCA International Conference on Computer Applications in Industry and Engineering (CAINE-2006), 13th-15th November 2006]. Las Vegas, Nevada, USA: International Society for Computers and Their Applications (ISCA), pp167-174.

This Conference Paper is brought to you for free and open access by the Computer Science at Glyndŵr University Research Online. It has been accepted for inclusion in Computing by an authorized administrator of Glyndŵr University Research Online. For more information, please contact d.jepson@glyndwr.ac.uk.

Complexity Issues in Control Software Design: A practical perspective

Abstract
There is a tendency to overlook or play down software issues in industrial systems design. However the very best hardware will only be as effective as the programs that control it. The search for a software solution to a problem should not stop with the discovery of the first approach that works, or appears to work. Many problems support good and bad methods of solution and determining the best often requires deeper consideration. In this paper we present some rudimentary introductions and examples of common computational challenges. We aim to highlight key issues for consideration when implementing control systems which otherwise might be unintentionally overlooked.

Keywords
Computational complexity, Control software, Algorithmic design, Optimization, Iteration, Recursion

Disciplines
Computer and Systems Architecture | Digital Communications and Networking | Hardware Systems | Systems and Communications

Comments
Copyright © 2006 ISCA and authors. This paper was presented at the19th ISCA International Conference on Computer Applications in Industry and Engineering (CAINE-2006), 13th-15th November 2006, which was held in Las Vegas, Nevada, USA. The conference organiser's website is located at http://www.isca-hq.org/ where a copy of the conference proceedings can also be purchased.

This conference paper is available at Glyndŵr University Research Online: http://epubs.glyndwr.ac.uk/cair/74

Complexity Issues in Control Software Design: A Practical Perspective

Vic Grout & Stuart Cunningham
Centre for Applied Internet Research (CAIR), University of Wales
NEWI, Plas Coch Campus, Mold Road, Wrexham, LL11 2AW, North Wales, UK
Tel: +44(0)1978 293583 Fax: +44(0)1978 293168
v.grout@newi.ac.uk | s.cunningham@newi.ac.uk

Abstract

There is a tendency to overlook or play down software issues in industrial systems design. However the very best hardware will only be as effective as the programs that control it. The search for a software solution to a problem should not stop with the discovery of the first approach that works, or appears to work. Many problems support good and bad methods of solution and determining the best often requires deeper consideration.

In this paper we present some rudimentary introductions and examples of common computational challenges. We aim to highlight key issues for consideration when implementing control systems which otherwise might be unintentionally overlooked.

1. Introduction

When we look at an industrial system, we see its hardware. The innovative technology and precision engineering are apparent. The years of research and subtleties of design are often implicit but equally obvious. Less tangible still (by definition) and often overlooked is the software that makes it all work. However without this software, of course, work it will not!

This might appear a trivial observation, sufficient to irk a software engineer maybe but nothing more. However to take on a design project with such a dismissive view of the software's importance will be to compromise the system's effectiveness from the outset. Worse, it may present obstacles to the success of the project that no amount of state-of-the-art hardware can overcome. Worse again, some things are not even worth trying!

This article offers a brief insight into some of the issues involved in problem-solving software design through two simple applications. Very simple in fact - the complexity of most industrial software systems would defy analysis in a few pages. Instead we look first at a generic form of problem which itself proves to be much too difficult with which to deal, then secondly at a cut-down version. Before that though, some basic principles are introduced.

2. The Limits of Algorithms

Actually some problems cannot be solved - at all that is. There are some problems for which it is impossible to produce a program that will deal with them. The most famous of these is Alan Turing's Halting Problem although there are many others. The Halting Problem asks for a program that will determine whether a second program will terminate naturally or become locked in an infinite loop. No such (first) program can exist. Although given examples of (second) programs can be dealt with individually, it is impossible to write a (first) program capable of pronouncing for all of them. We do not attempt to prove this statement here [1].

In laying aside the unsolvable problems to concentrate on the solvable, we immediately exhaust the typical mathematician's interest in the subject. However, among solvable problems, there appear to some that are essentially harder than others. The study of such differences has occupied the computational mathematician for the last two or three decades and some questions remain unanswered. That which is understood becomes the realm of the algorithmic designer. An algorithm is simply a program in abstract form - that is, independent of language or implementation.

```
// Skeleton program - 1D loop
  :
main() {
  :
 for (i = 1; i <= n; ++i) {
  :
  }
  :
}
```

Figure 1 - A One-Dimensional Loop

```
// Skeleton program - 3D loop
   :
main() {
   :
   for (i = 1; i <= n; ++i) {
      :
      for (j = 1; j <= n; ++j) {
         :
         for (k = 1; k <= n; ++k) {
            :
         }
         :
      }
      :
   }
   :
}
```

Figure 2 - A Three-Dimensional Loop

Consider the simple code fragment in Figure 1. Here a single loop runs a control variable from *1* to some value *n*. If *n* is the key value of the program, the number of data items to be processed, say, and there are no more significant loops, then the complexity of the program can be seen to increase linearly with *n*. In fact we can express this complexity as $O(n)$. There is no need to consider the number of instructions within the loop or the number of similar loops. In Figure 2, by the same process, the complexity can be written as $O(n^3)$, no longer linear in *n* but polynomial. Any program having a complexity expression of the form $O(n^x)$ can be described as having polynomial complexity. Lesser exponents may be ignored: $O(n^4 + n^2)$, for example, can be simplified to $O(n^4)$ since, for increasing values of *n*, n^4 becomes the dominant term. The complexity of a program is effectively a measure of how much it will begin to slow down as *n* increases. Programs with polynomial complexity of $O(n^x)$ will slow down more for larger values of *x* as illustrated in Figure 3. However, there are worse cases to consider.

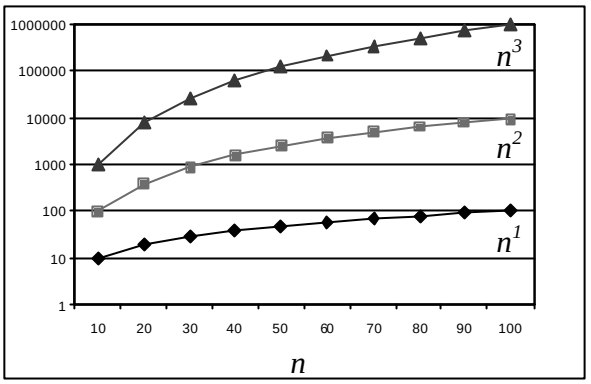
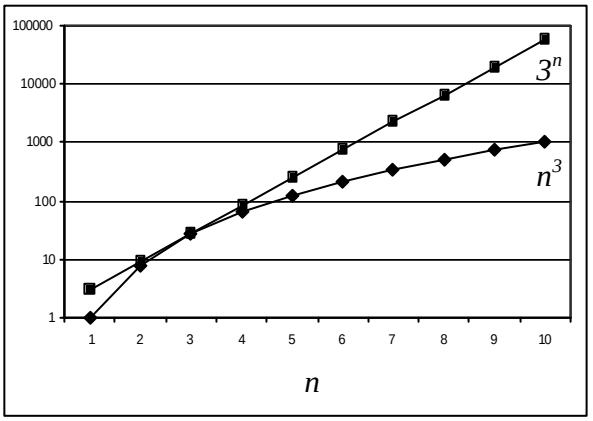


Figure 3 – Various n^x Curves
Figure 4 – Various n^3 and 3^n Curves

Some programs will have greater than polynomial complexity – quite how, we shall see in the next two sections. The common term for this is *exponential* complexity. This is a slight generalisation but will serve us well enough. $O(x^n)$ is the generic form of exponential complexity. Figure 4, for example, shows how 3^n increases compared to n^3 as *n* increases. In simple terms, programs of polynomial complexity are generally acceptable - those of exponential complexity, not.



These descriptions of program complexity can be equally applied to the problems they are attempting to solve providing, that is, that we can be sure we have the most efficient program for a given problem. As we shall see later, there are good and bad ways of writing programs to solve the same problem. To be able to accurately classify a problem we need to know the best algorithm.

Many problems, and most 'every-day' ones, are known to have polynomial complexity. These are collectively identified as the class *P*. The wider class of problems having polynomial or exponential complexity is generally

referred to as *NP*. Technically, *NP* stands for 'non-deterministic polynomial' rather than 'non-polynomial', as might be supposed, but that need not concern us here. Clearly *P* is a subset of *NP* as shown in Figure 5.

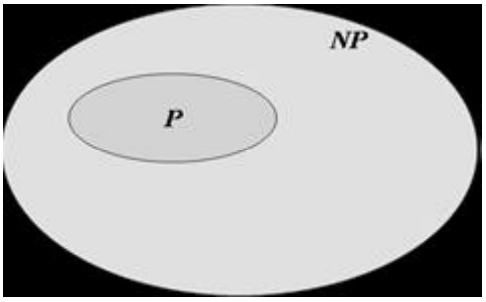


Figure 5 – The Classes *P* and *NP*

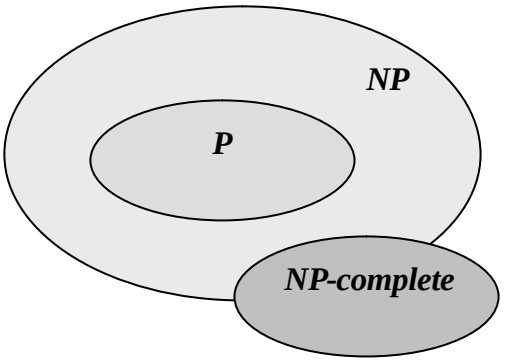
Figure 6 – The Classes P, NP and NP-Complete?

A reasonable question would be to ask if there really are any problems in *NP* that are not in *P* or whether they are actually the same class. In other words, are there any problems that are genuinely exponentially complex or can all problems be solved by a polynomial algorithm if only we can find it? Are exponentially complex algorithms necessary in some cases or simply the result of poor design?

Essentially, the answer to this question is unknown. The current situation is summarised as follows:

1. There are many problems for which polynomial algorithms are known.
2. There are many problems for which polynomial algorithms are not known.
3. There are a number of these problems, for which polynomial algorithms are not known, which are known to be equivalent to each other in the sense that the discovery of a polynomial algorithm for one would effectively provide polynomial algorithms for all. This class of problems is referred to as *NP-complete*.
4. It has not been proved that no polynomial algorithm exists for any *NP-complete* problem.

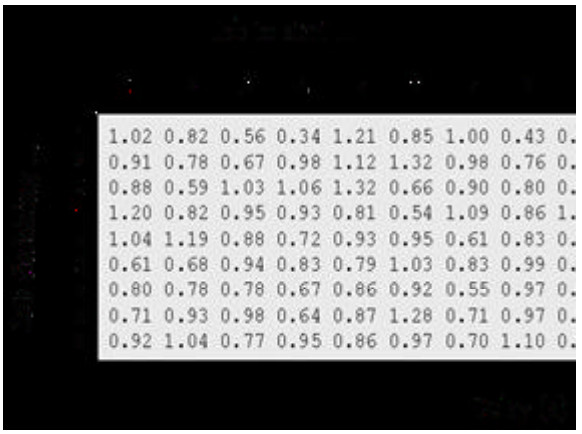
Points 1, 2 and 3 suggest a relationship between the classes as shown in Figure 6. However, point 4 means we cannot be sure. The discovery of a proof that *NP-complete* problems can not be solved by polynomial algorithms would dispel all doubt. The discovery of such a polynomial algorithm, however, would be unexpected to say the least.



3. A Job-Sequencing Problem

All of this may sound a little theoretical, lacking practicality, so we introduce an example. Consider a piece of machinery, able to perform a number of processes. Suppose, in general operation, that this multi-functional device is to perform *n* tasks, each of which involves the machine starting and finishing in a number of different positions or settings. If we represent the time taken to reset the machine to start job *j* after finishing job *i* by t_{ij} then a complete description of the delays involved is given by the matrix $T = (t_{ij})\ 1 \pounds i,j \pounds n$, an example of which is given in Figure 7.

Figure 7 – A Delay Matrix Example



1.02	0.82	0.56	0.34	1.21	0.85	1.00	0.43	0.
0.91	0.78	0.67	0.98	1.12	1.32	0.98	0.76	0.
0.88	0.59	1.03	1.06	1.32	0.66	0.90	0.80	0.
1.20	0.82	0.95	0.93	0.81	0.54	1.09	0.86	1.
1.04	1.19	0.88	0.72	0.93	0.95	0.61	0.83	0.
0.61	0.68	0.94	0.83	0.79	1.03	0.83	0.99	0.
0.80	0.78	0.78	0.67	0.86	0.92	0.55	0.97	0.
0.71	0.93	0.98	0.64	0.87	1.28	0.71	0.97	0.
0.92	1.04	0.77	0.95	0.86	0.97	0.70	1.10	0.

Now consider the software controlling this device. A useful feature would be for the system to automatically calculate in which order the jobs should best be executed. This optimum would be an ordering of the jobs *1, 2, ... i, j, ..., n* such that the sum of the t_{ij}s is minimised. If there is more than one round of jobs to process then it will be necessary to reset the machine to start the first job after the last each time and the problem becomes cyclic.

This question, often rephrased as the Travelling Salesman Problem or TSP is known to be *NP-complete* [2, 3]. Considering the previous section, this sounds unpromising but what does it actually mean in practice? Is it going to be difficult to write a program for the task?

In fact, writing the program, a crude one at least, can be fairly routine. The obvious solution will be to generate each permutation of the *n* jobs in turn, calculate the total delay for each and progressively record the best. There are a number of approaches available: Knuth [4] outlines a recursive method; a more mundane iterative process is offered here.

If we plan to begin at a known starting point (the 'lowest' permutation, *1,2,3,...,n*, being the natural choice) and continue until a recognisable end is encountered (again, the 'highest' permutation, *n,...,3,2,1*, the obvious candidate) then, save fo r the trivial task of evaluating each permutation against the delay matrix, the only requirement is for a routine to successively generate each permutation from its predecessor. Figure 8 gives such a function, used within a harness program to simply output each permutation rather that applying the delay matrix. Its principles are outlined in Figure 9.

```
// Generating permutations of integers

#include <iostream.h>

int i,j,k, n, a[10], b[10], done = 0;
long int count;

main() {

   cout << "Enter n: ";
   cin >> n;
   for (i = 0; i < n; ++i)
      a[i] = i + 1;              // Initialise array: 1,2,3,..,n

   do {
      cout << ++count << "   ";  // Output count
      for (i = 0; i < n; ++i)
         cout << " " << a[i];    // Output permutation
      cout << "\n";

      if (n == 1)
         done = 1;               // Only one permutation
      else {                     // Generate next
         i = n-2;
         while (a[i] > a[i+1] && i >= 0)
            --i;                // Find focus (i)
         if (i < 0)
            done = 1;           // Last permutation
         else {
            j = i + 1;
  for (k = i + 2; k < n; ++k)
     if (a[k] > a[i] && a[k] < a[j])
        j = k;         // Find new focus (j)
  k = a[i];
  a[i] = a[j];         // Swap foci (i, j)
  a[j] = k;
  for (k = i + 1; k < n; ++k)
     b[k] = a[k];      // Copy rest to dummy
  for (k = i + 1; k < n; ++k)
     a[k] = b[n-k+i];  // Read back in reverse
         }
      }

   } while (!done);
}
```

Figure 8 – Generating Permutations

Figure 9 – Iterating Permutations

Working from right to left through the existing permutation, all pairs of values with left larger than right will represent permutations already considered. Scope for a new ordering arises when a pair is encountered in which right is larger than left. The position of this left value becomes the focus for the rearrangement to follow. Of all values to the right of the focus, the minimum value larger than the current focus is selected and the two are swapped. All values to the right of the new focus are now reversed into ascending order.

This is not an elegant program – it has been written for efficiency – and we expand upon this in the next section. In fact however, the problem that arises is not to do with the logical difficulty of the program or the performance of its main function, but its search complexity. How many permutations is it to generate? This is simple to calculate. The first job may be chosen in n ways, the second in $n-1$, the third in $n-2$ and so on. With two jobs left to choose between there are 2 choices, which leaves a single choice for the last. The number of possible permutations is therefore

$$p(n) = n \times n-1 \times n-2 \times \ldots \times 2 \times 1 = n! \quad (1)$$

which is clearly exponential in form. Figure 10 shows the first few values plotted against 3^n for comparison. It is tempting to think that modern computing power will make short work of this but a moment's consideration suggests otherwise.

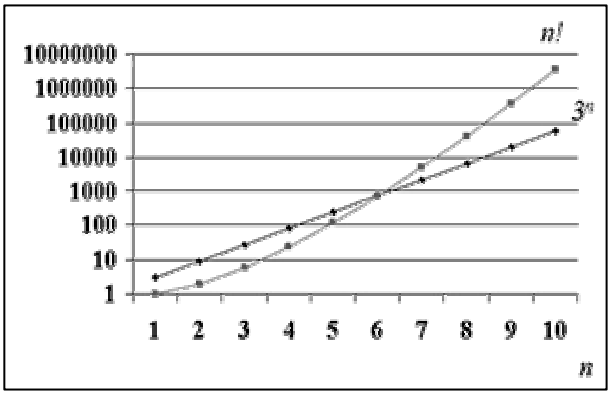


Figure 10 – The growth of *n!*

Suppose, for the sake of argument, that, with whatever processor we have available, we can use the routine in Figure 8 in a program to solve the job-sequencing problem for $n = 20$ maximum. Consider what is necessary to introduce an extra job.

$$p(21) = 21! = 21 \times 20! = 21p(20) \quad (2)$$

We will need an increase in processing of 21 times to deal with the single extra job and $22 \times 21 = 462$ times to accommodate another two. To solve the problem for $n = 30$ jobs, $109{,}027{,}350{,}432{,}000$ times as much power is required! The problem is genuinely hard! Whilst it is true that there are slightly better methods that this simple exhaustive search approach [2, 3], there are none known that bring the complexity down to polynomial levels. The options are limited: avoid the problem, deal with it on a small scale only or keep searching for an algorithm that probably does not exist!

4. Another Job-Sequencing Problem

However, there is another, more down-to-earth, aspect of algorithmic design. Even for easy problems, it is quite possible to write bad programs. For any given polynomial problem, there may be a variety of algorithms available. Very poor design may lead to exponential algorithms and even among the polynomial options, selecting the best may not be a trivial exercise. Efficiency and inefficiency are often disguised. A second example will serve to illustrate a number of these points.

In fact, a variation of the previous example - a simplified one - will serve us well. Suppose that the machine in question can perform only two jobs and that the delay between them is no longer a concern. Suppose, however, that job 1 takes 1 second to process, job 2, 2 seconds and that the machine can run for n seconds before being reset

for a new run. To maximise machine use requires a sequence of jobs *1* and *2* of duration *n*. How many such sequences are there?

Admittedly, it is not obvious why it might be necessary to calculate this figure - the machine would function well enough without it. However it offers us a manageable problem to study. If *f(n)* represents the number of different sequences of jobs *1* and *2* totalling *n*, how can we calculate *f(n)*?

The first values are trivial. For $n = 1$, there is only one possibility - a single job *1*, so $f(1) = 1$. For $n = 2$, we could run job *1* twice or job *2* once: $f(2) = 2$. To derive a general expression for *f(n)* requires a little analysis.

Having dealt with $n = 1$ and $n = 2$, suppose $n \geq 3$ and consider the first job. Job *1* leaves *n-1* seconds to fill, which can be done in *f(n-1)* ways and job *2* leaves *n-2* seconds and *f(n-2)* ways. This gives us the relation

$$f(n) = f(n-1) + f(n-2) \quad (3)$$

which is recognisable as the Fibonacci Sequence in which each pair of numbers is added to obtain the next. A simple enough expression, it remains only to turn it into a program. However, without due care, this may be the start of our problems.

There are probably two particularly obvious solutions. Firstly, (3) in programming terms is a recursive definition and as such can be coded directly. Figure 11 gives a short program using this method. Secondly, the formula may be applied in reverse with each new term being iterated from the two before. Figure 12 uses this approach. Both programs attempt to calculate *f(n)* for values of *n* from *3* to *100*, the values of *f(1)* and *f(2)* being trivially known.

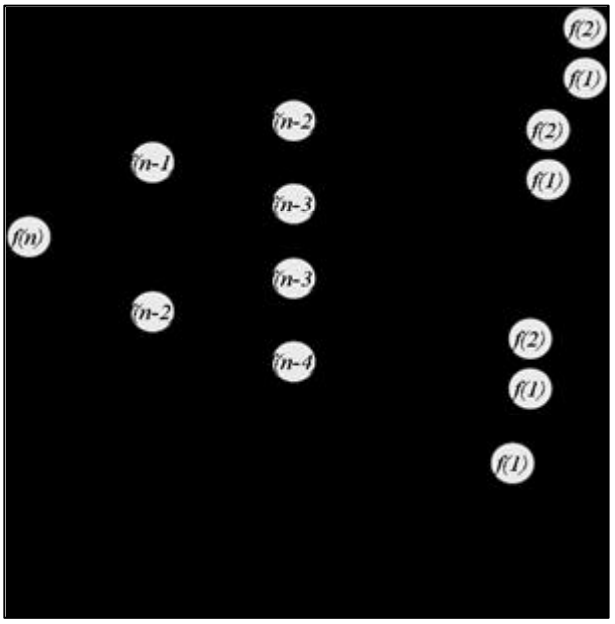
```
// Calculating f(n) by recursion

#include <iostream.h>

int n;
float f(float);

main() {
   for (n = 3; n <= 100; ++n)
      cout << "n = " << n << ":  f(n) = " <<
f(n) << "\n";
}

float f(float n) {
   if (n ==1 || n ==2)    // Trivial values
      return n;
   else
         return f(n-1) + f(n-2);
        // Recursive call
}
```

Figure 11 – Calculating *f(n)* Recursively

```
// Calculating f(n) by iteration

#include <iostream.h>

int n;
float fn, fn_1 = 2, fn_2 = 1;
// Trivial values

main() {
   for (n = 3; n <= 100; ++n) {
      fn = fn_1 + fn_2; fn_2 = fn_1; fn_1 =
fn;  // Iteration
      cout << "n = " << n << ":  f(n) = " <<
fn << "\n";
   }
}
```

Figure 12 – Calculating *f(n)* Iteratively

Although similar in size, the performance of the two programs could not be more different! While the iterative program in Figure 12 runs to completion almost



instantaneously, the recursive method in Figure 11, after initial pace, reaches a point where it slows visibly and continues to slow until it practically stops altogether. Each new calculation takes longer than the last.

Figure 13 – Complexity of calculating *f(n)* Recursively

Figure 14 – Complexity of calculating *f(n)* Iteratively

This behaviour is explained in Figures 13 and 14. The iterative method has polynomial (in fact linear) complexity. To derive each successive value requires only a single step; its complexity is $O(n)$. The recursive method, on the other hand, calculates each value via two calls to the same function which, in turn, makes two more calls, etc. Its complexity is approximately $O(2^n)$: it is exponential.

There is no doubt which of these methods is the better. However, there is another, less obvious approach. A mathematician would call (3) a recurrence relation and such expressions can often be resolved to give explicit values for $f(n)$. Noting that certain types of relation give predictable forms of solution and substituting the known initial values of $f(1)$ and $f(2)$, we can derive the expression [5]

$$f(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right] \quad (4)$$

which suggests a method of solution with no repetition at all. The program in Figure 15 implements this method using standard library functions and, on the surface, appears efficient.

```
// Calculating f(n) in one step?

#include <iostream.h>
#include <math.h>

int n;

main() {
   for (n = 3; n <= 100; ++n)
      cout << "n = " << n << ":  f(n) = " <<
         (pow((1 + sqrt(5)) / 2, n+1)
        - pow((1 - sqrt(5)) / 2, n+1)) /
sqrt(5) << "\n";
}
```

Figure 15 – Calculating *f(n)* in one step?

Unfortunately, there are some hidden dangers here. These powerful high-level operations will ultimately have to be translated into simple machine code to be executed. At this level, the calculations of powers and roots are non-trivial. Implementations vary but the square-root operation, for example, is likely to be calculated by numerical approximation methods, successively generating improved versions of the result until the required accuracy is obtained. The apparently banished iteration returns! Although the difference will not show in these examples, this method of solution proves to be inferior to that of Figure 12.

```
// Calculating f(n) by analysis

#include <iostream.h>
#include <math.h>

int n;
float root5 = sqrt(5),
      term1, factor1 = (1 + root5) / 2,
      term2, factor2 = (1 - root5) / 2;

main() {
   term1 = factor1 * factor1 * factor1;
   term2 = factor2 * factor2 * factor2;
// Initialise
   for (n = 3; n <= 100; ++n) {
      term1 *= factor1; term2 *= factor2;
// Iterate
      cout << "n = " << n << ":  f(n) = "
           << (term1 - term2) / root5 <<
"\n";
   }
}
```

Figure 16 – Calculating *f(n)* by analysis

However, using the information from (4) wisely, can make a difference. Looking carefully, we see that only integer powers are needed, the same exponents for each term and increasing in line with (or one in advance of) n. Also, only a single root value is ever required. The program in Figure 16 exploits these features by calculating Ö5 once at the start, subsequently using the calculated value, and generating each power term from the previous one. It is not superior to the method of Figure 12 (and it is certainly not elegant) but it is comparable.

5. Summary

A number of points should be highlighted in conclusion. First and foremost, presumably, is the fact that the status of some of these complexity classes is unknown - and may well remain so. In fact, unless a 'magic' polynomial for an *NP-complete* problem is found, this has little practical effect. Complexity theorists generally accept the relationship in Figure 6 and algorithmic designers are all too familiar with the line between polynomial and exponential problems and algorithms. Making the best of a bad lot is the general order.

The first job-sequencing example shows that it is not difficult to write a program for a problem that cannot realistically be solved! The hitch is that the program will never finish for larger input values. A different point is worth making in conclusion. The program in Figure 6 could be made considerably more readable by structuring it with more functions. In fact, according to the principles of good design, this was indeed the form in which it was originally developed. However, removing subroutine calls and parameter passing from the underlying implementation improves efficiency substantially. The resultant single-block program is not easy to follow and would not please the purists but it is a 'better' program from our point of view. Of course, due to the exponential nature of the problem, it has an insignificant overall effect of the size of input with which we are able to deal but it could make a useful difference to the limit value. In the drive for program speed, elegance and efficiency rarely coexist!

The second job-sequencing example introduces some similar lessons. Careless design can produce a complex program for a simple problem. More specifically, recursion as a programming tool is often elegant, rarely efficient - or at least, rarely more efficient. In this case its use is disastrous. We also need to be wary of apparent 'miracle' solutions. They may not be all they seem. However, a close look at a problem can reveal a trick or two, sometimes more functional than aesthetically pleasing. Again the 'artistic' programmer and the efficiency seeker often steer different courses.

6. References

[1] Cutland, N.J., *Computability: An Introduction to Recursive Function Theory*, Cambridge University Press, 1980.

[2] Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G. & Shmoys, D.B., *The Travelling Salesman Problem*, Wiley, 1985.

[3] Gutin, G., Punnen, A.P., *The Travelling Salesman Problem and its Variations*, Kluwer Academic Publishers, 2002.

[4] Knuth, D.E., *The Art of Computer Programming*, Vol III: Sorting and Searching Addison Wesley, 1998.

[5] Anderson, I., *A First Course in Combinatorial Mathematics*, Clarendon Press, 1989.