Basic Science for Software Developers

David Lorge Parnas, P.Eng.
Michael Soltys
Department of Computing and Software
Faculty of Engineering
McMaster University, Hamilton, Ontario, Canada - L8S 4K1

1 Introduction

Every Engineer must understand the properties of the materials that they use. Whether it be concrete, steel, or electronic components, the materials available are limited in their capabilities and an Engineer cannot be sure that a product is “fit for use” unless those limitations are known and have been taken into consideration. The properties of physical products can be divided into two classes:

- *technological properties*, such as rigidity, which apply to specific products and will change with new developments,
- *fundamental properties*, such as Maxwell’s laws or Newton’s laws. These properties will not change with improved technology.

In many cases technological properties are expressed in terms of numerical parameters and the parameter values appear in product descriptions. This makes these limitations concrete and meaningful to pragmatic developers. It is the responsibility of engineering educators to make sure that our students understand the technological properties, know how to express them, know how to determine them for any specific product, and know how to take them into account when designing or evaluating a product.

However, it is also the responsibility of engineering educators to make sure that our students understand the fundamental limitations on the materials that they use. It is for this reason, that accredited engineering programmes are required to include a specified amount of basic science. Explaining the relevance of basic science to Engineers is a difficult job; Technological limitations are used to compare products; in contrast, fundamental limitations are never mentioned in comparisons because they apply to all competing products. As a result, the technological limitations seem more real and students do not perceive fundamental limitations as relevant because they do not expect to have to apply their understanding of those limitations. Nonetheless, an understanding of basic science is essential to an Engineer. For example, an Engineer who understands the principle of conservation of energy can often find quick and simple solutions to problems that appear difficult to others.

For Software Engineers, the materials used for construction are computers and software. In this area too, the limitations can also be divided into two classes:

- *technological limitations*, such as memory capacity, processor speed, word length, types of bus connections, precision obtained by a specific program, availability of specific software packages, etc.,
- *fundamental limitations*, such as limits on computability, complexity of problems, and the inevitability of noise and other forms of error in data.

Computer Scientists have developed a variety of useful models that allow us to classify problems and determine which problems can be solved by specific classes of computing devices.

The most limited class of machine is the finite state machine. Finite state machines can be enhanced by adding a “last-in-first-out” memory known as a stack. Adding an infinitely extensible tape that can move both forwards and backwards through the reader/writer makes the machine more powerful (in an important sense) than any computer that can actually be built. Practising software developers can use these models to determine how to approach a problem. For example, there are many problems that can be solved completely with the simplest model, but others must be restricted before they can be solved. Many people know these things in theory, but most do not understand how to use the theory in practice.

Like the students in other engineering disciplines, software engineering students must be able to understand and deal with the technological limitations. Even the youngest have seen rapid improvements in technology and understand how to characterise what has changed. They see that products differ in these properties and most of them easily understand the practical implications of those differences.

It is not useful to spend a lot of time on the technological limitations of specific current products. Much of what students learn about today’s products will be irrelevant before they graduate. However, it is very important to teach the full meaning of technological parameters and how to determine which products will be appropriate for a given application.

The fundamental laws that professional software developers should understand limit the capabilities of *all* products. Because these limitations do not distinguish products, they are not mentioned in product descriptions and are rarely well understood by software practitioners. Consequently they often seem irrelevant to students, some software educators, and most experienced developers.

Nonetheless, the fundamental properties of computers are very important because they affect what we can and cannot do. Sometimes, an understanding of these properties is necessary to find the best solution to a problem. In most cases, those who understand computing fundamentals can anticipate problems and adjust their goals so that they can get the real job done. Those who do not understand these limitations, may waste their time attempting something impossible or, even worse, produce a product with poorly understood or unclearly stated capabilities. Further, those that understand the fundamental limitations are better equipped to clearly state the capabilities and limitations of a product that they produce. Finally, an understanding of these limitations, *and the way that they are proven*, often reveals practical solutions to practical problems. Consequently, this “basic science” should be a required component of any accredited Software Engineering programme.

In the next section, we will give a few illustrations to make these points clearer. In the final section, we will sketch a course designed to introduce Software Engineering students to the basic science that is specific to their discipline.

2 A few anecdotes

2.1 What can be said with grammars

Many years ago, Robert Floyd encountered a graduate student who was trying to find a complete context-free grammar for Algol-60, one that specified that all variables must be declared before use. The student’s plan was to use the grammar as input to a compiler generator. Floyd’s understanding of CS fundamentals allowed him to prove that no such grammar could exist. The graduate student was saved months, perhaps years, of futile effort. With this information he understood that he would have to find another way to express those restrictions as input to his compiler generator. [1]

This anecdote makes it clear that it is very important to be able to decide whether or not a task is impossible. Some people spend their lives trying to solve impossible problems.

2.2 Arithmetic Decisions

A problem is decidable if we can come up with an algorithm to solve it. More precisely, given any input, our algorithm must compute the correct output, and it must always halt.

For example, the Satisfiability problem [7,9] is decidable: given a boolean formula, we can check if there is a truth value assignment to its variables that makes the formula true. The Satisfiability problem can be solved with the following brute force algorithm: check systematically all possible truth value assignments to the variables in the formula. For n variables, there are 2^n assignments, so we will either find one that satisfies it, or check them all and conclude that the formula is unsatisfiable.

Note that to show that a problem is decidable, we must show that there exists an algorithm that solves it and always halts. On the other hand, to show that a problem is undecideable, we must show that there is no algorithm that solves it. Thus, showing undecidability involves showing that any algorithm that we apply to our problem fails to provide correct answers (or any answer at all, if it fails to halt) on some inputs (note that if it fails, it has to fail on infinitely many inputs, since otherwise we could “patch it up” on those finitely many” bad” inputs).

The Halting Problem is the prototypical example of an undecideable problem. The truth of statements about arithmetic is another such example [9].

Interestingly enough, if we restrict arithmetic to the usual statements, but without multiplication (for example, statements like: a+b=b+a, or a+(b+c)=(a+b)+c), then this fragment of arithmetic, called Presburger’s arithmetic, is decidable!

Sometimes, a small modification or restriction makes an otherwise impossible problem tractable. For example, while it is true that some problems in number theory are undecideable [8], those problems in a number theory with only “+” (Presburger's Arithmetic) are decidable. It is true the computational complexity of the algorithm is very high, but in some cases, this is not an insurmountable barrier. In other words, if it is not the most general case of a given problem that we have to solve, negative results may not be applicable, but the practitioner must understand why.

2.3 The meaning of computational complexity.

Computer Scientists have developed ways to classify the complexity of algorithms and to classify problems in terms of the complexity-class of the best solution to those problems. This allows them to determine whether or not an algorithm is a good as it can get (optimal). However, strange as it may sound, sometimes an “optimal” algorithm is not the best choice for a practical application.

In the 70’s Fred Brooks, working on visualisation tools for chemists, announced that he wanted an optimal algorithm for a well defined problem. A very bright graduate student proposed such an algorithm and submitted a short paper proving that it was optimal. Brooks insisted that the algorithm be implemented and its performance compared with the performance of the method that they had been using; the performance of the “optimal” algorithm was worse than the old one. Computer Science complexity methods refer to asymptotic performance, that is performance for very large problems. Algorithms that are not optimal may actually be faster than the “optimal” ones for certain values of the key parameters. Since a developer may find an “optimal” algorithm in a textbook, she must be aware of what “optimal” means and check to see that the performance is actually better in practice than other algorithms. Moreover, a developer who knows the asymptotically optimal algorithm can often modify it to produce an algorithm that will be fast for the application at hand. Another such example is Linear Programming. A widely used algorithm is the Simplex Method, known to be exponential in the worst case. However, the Simplex Method ha superb performance in practice (in fact, it’s expected performance is provably polynomial). On the other hand, the first “worst-case” analysis polynomial algorithm for Linear Programming, known as the Ellipsoid Algorithm, appears to be impractically slow. [9]

Similarly, the fact that a problem is known to belong to a high complexity class (NP-complete), does not mean that we cannot do anything with it. We may be able to solve it for small values of the parameters using brute force, we may be able to use approximation algorithms, or we may be able to restrict the problem to a special case for which we can use a less complex algorithm. For example, while 3-colourability [7] is NP-hard, 2 colourability has a simple polytime algorithm, or while 3-SAT is NP-hard, 2-SAT has a simple polytime algorithm.

Further, while general boolean expression in conjunctive normal form are hard to decide if they are satisfiable, a large class of them, called Horn Formulas [9], have a simple polytime algorithm for satisfiability (a fact that has been exploited by researchers in Artificial Intelligence). Here too, an understanding of computational complexity, can lead a developer to a practical algorithm that they might not otherwise find.

2.4 The practicality of a “bad” solution to the “Knapsack Problem”

In the “Knapsack Problem” the input is a set of weights w1,w2,...,wd, and the capacity, C, of the knapsack. We want to pack as much weight into the knapsack, without going over the capacity.

The most obvious approach, starting with the largest weights, does not work, because if we have three weights w1=51,w2=50,w3=50, and C=100, and our strategy is to pack as much as possible at each step, we would put 51 in, and we would have no more space left for w2 and w3. The optimal solution is of course w2+w3=100.

The “Knapsack Problem” can be solved with Dynamic Programming, where we construct a table with dimensions (d, C) (d= number of weights, C=capacity), and fill it out using simple recursion. A classical worst-case running time analysis of the dynamic programming algorithm shows that it requires exponential time. The reason is that the algorithm builds a dxC table, so if C is given in binary, the size of the table in exponential in the size of the capacity (i.e., exponential in the size of the input). Therefore, the dynamic programming solution to the Knapsack Problem runs in exponential time in the size of the capacity of the knapsack, and hence it is asymptotically infeasible.

In fact, the dynamic programming solution to the “Knapsack Problem” is ubiquitous in Computer Science. In applications such as Compilers and Optimization problems, equivalent problems arise frequently, and they are solved using dynamic programming. The method is practical, even with many weights, for reasonable C.

Thus, even though “Knapsack Problem” is in a bad complexity class, and the dynamic programming solution is exponential in the capacity (C), it is nevertheless a very good solution for many situations.

One should not interpret this as meaning that the theoretical complexity is useless; *au contraire,* it demonstrates why even practitioners who think that they are not interested in “theory” should understand computational complexity when developing algorithms for difficult problems.

2.5 Maximum size for a halting problem

Two software developers were asked to produce a tool to check for termination of programs in a special purpose language used for building discrete event control systems. One refused the job claiming that it was impossible because we cannot solve the halting problem. A second, who understood the proof of the impossibility of the halting problem, realised that the language in question was so limited that a check would be possible if a few restrictions were added to the language. The resulting tool was very useful. Here again, an understanding of the nature of this “very theoretical” result was helpful in developing a practical tool with precisely defined limitations.

2.6 Can we prove that loops terminate

Dr. Robert Baber, a software engineering educator who has long advocated more rigorous software development [2, 3, 4, 5] was giving a seminar in which he stated that it was the responsibility of programmers to determine whether or not loops they have written will terminate. He was interrupted by a young faculty member who asserted that this was impossible, “the halting problem says that we cannot do that”. In fact, the halting problem limits our ability to write a program that will test *all* programs for termination, not about our ability to check a given program for termination or the importance of writing programs in such a way that checking for termination is possible and carried out rigorously. This incident shows that a superficial understanding of computer science theory can lead people astray and cause them to be negligent.

Clearly, we must teach fundamentals in such a way that the student knows how to translate theoretical results into practical knowledge. For example, when teaching about the general undecidability of halting problems, one can accompany the proof with an assignment to determine the conditions under which a particular machine or program is sure to terminate. Comparing the general result with the specific example helps the student to understand the real meaning of the general result.

2.7 The implications of finite word length.

In 1969, some software developers became enthusiastic about a plan to store 6 bytes in a 4 byte word. They proposed computing the product of 6 bytes and converting the result to a 4 byte floating-point number. Sadly, none of the programmers in the organization understood the impossibility of this scheme and they invested a lot of time discussing it. Luckily, an academic visitor who did understand basic information theory, could convince them of its applicability by providing a counter-example, i.e an example where the same output would be obtained for two different inputs. It was quite possible that even extensive testing would not have revealed the error, but it would cause “bugs” in practice.

2.8 The limitations on push-down automata.

Recently one of us had occasion to talk to some people who were familiar with the standard results about push-down automata, i.e. that the class of problems that they could solve was smaller than that for Turing machines. He reminded them that in today’s market, one can buy an auxiliary disk and attach it to a lap top or other personal computer. He asked if this changed the fundamental properties of the machine (it does not). He then asked what would happen if we could buy an auxiliary push-down stack and attach it as a separate device on a push-down automata that already had one stack. All claimed that the result would still be a push-down automata, i.e. they did not recognize that having a second (independently accessible) stack changed the fundamental capabilities of the machine. The same group included many who did not realise that placing limits on the depth and item size of the stack in a push-down automaton made it no more powerful than any other finite state machine. This meant that they did not understand that there would be an upper limit in the number of nested parenthesis in an expression that would be parsed by any realisable push-down automaton, or that a twin-stack push-down automaton (with infinite stacks) was as powerful as a Turing machine and more powerful than any realisable computer.

2.9 The practical limitations of open questions.

There are number of problems in computability and complexity theory that remain open. Many practitioners and students believe that these problems are of interest only to theoreticians. In fact, they have very practical implications. Probably the most dramatic of these is the "P = NP" question [6] for which a prize of $1,000,000 has been offered. This does not interest most students who realize that they will not win the prize. However, the question has very important implications in cryptography. Some very widely used encoding algorithms are only "safe" if the answer is that P $\neq$ NP. If it is not, it might be possible to find ways to crack codes quickly [6, 7].

3 A course in basic science for Software Engineers.

McMaster Universities CEAB accredited Software Engineering Programme includes a course designed to teach its students both the parts of "theoretical computer science" that they ought to know and how to use them. This section sketches the contents of that course. More complete descriptions are available on the appropriate web pages. Deeper discussions can be found in [7, 8, 9].

1. Finite Automata (finite number of states, and no additional memory)
   - deterministic finite automata (DFA)
   - nondeterministic finite automata (NFA) (Given any state and an input, there may be more than one choice for the next state.)
   - nondeterministic finite automata with "epsilon" transitions (epsilon-NFA) (An epsilon transition is a spontaneous transitions, that is, a state transition that occurs when there is no input.)
   - regular languages
   - DFAs, NFAs, and epsilon-NFAs, recognize the same classes of languages; that is, whatever can be computed with one of them, can be computed with the others
   - Pumping Lemma for showing that certain languages are not regular (in general, showing that a problem cannot be solved with a given model of computation, or within a given complexity class, is very difficult, so the Pumping Lemma is nice, as it is a simple and easy to use tool for proving that particular languages cannot be decided with finite automata).

2. Regular Expressions (the basis of many useful notations and compiler tools)
   - building regular expressions
   - converting DFAs to regular expressions
   - converting regular expressions to epsilon-NFS (thus finite automata and regular expressions are equivalent)
   - regular expressions in UNIX

- algebraic rules for regular expressions
- closure properties of regular expressions (complements, intersections, unions)
- testing emptiness and membership in regular expressions (this is undecideable for Turing Machines, but decidable for regular expressions)
- equivalence of automata (algorithm for showing that two DFAs are equivalent)
- minimal automata (algorithm for producing a minimal DFA equivalent to a given DFA)

3. Context-Free Grammars - (used to describe many programming languages)
   - derivations (show that a given string is described by a given CFG)
   - left-most and right-most derivation
   - parse trees
   - trees and derivations
   - YACC parser generator (also mention XML)
   - ambiguity
4. Pushdown Automata (a useful model for many algorithms in language processing)
   - equivalence of PDA and CFG
   - PDA with two stacks are equivalent to general computers
   - PDA with finite stack are equivalent to finite automata
5. Turing Machines (TM) (simplified model of a general computer, but more powerful than real computers)
   - Design of TM
   - encoding a TM as a string of symbols
   - languages vs problems (i.e., the mapping between classes of languages and classes of problems)
   - Church-Turing thesis (all algorithms can be simulated on TMs)
   - Robustness (all variants of TMs are equivalent) (many tapes, tapes infinite in one direction only)
   - Decidable, Undecideable, and Semi-decidable languages
   - Halting Problem (can we design an algorithm, which given a TM and an input to that TM, decides if the TM will halt; answer: NO)
   - Diagonalization (a method for showing that problems like the Halting Problem is not decidable)
   - Enumerable languages, are languages whose elements can be listed (enumerable languages are not necessarily decidable languages, since at any giving point in the listing of the elements, we do not know if the element we are waiting for is not on the list, or simply has not appeared yet). Enumerable languages are equivalent to semi-decidable languages.
6. Rudimentary Computational Complexity
   - Feasibility Thesis (TMs are reasonable model of computation; a polytime program on a RAM machine requires polynomially many steps on a TM)
   - the class P, the class NP (P: problems which can be solved in polytime, and therefore, feasibly. NP: problems whose solutions can be verified feasibly)
   - the P vs NP problem, relation to feasibility, and relation to cryptography (cryptography secure if P not= NP)
   - P vs NP: verifying vs computing

- NP-completeness (if SAT is in P, P=NP)
- reducing practical problems to known problems (e.g. all NP-complete problems are really SAT in disguise)

4 Conclusions

Established engineering accreditation rules require that each engineering student have a minimum exposure to basic science. As accredited software engineering programs are relatively new, there is no clear understanding what constitutes appropriate basic science. Although we believe that every Engineer should have been taught basic physical science, we believe that those who will specialise in software require a thorough exposure to the topics discussed above. This paper has illustrated why, a course on these topics should be required as part of the basic science component of a programme for engineers specialising in software intensive products.

5 References

1. Robert Floyd, personal communication.
2. Baber, R.L. “Software Reflected: the Socially Responsible Programming of Our Computers”, North-Holland Publishing Co., Amsterdam, 1982. German translation: Softwarereflexionen: Ideen und Konzepte für die Praxis, Springer-Verlag, 1986.
3. Baber, R.L**, “**The Spine of Software: Designing Provably Correct Software - Theory and Practice”, John Wiley & Sons, Chichester, 1987.
4. Baber, R.L., “Error Free Software: Know-How and Know-Why of Program Correctness”, John Wiley & Sons, Chichester, 1991. German original: Fehlerfreie Programmierung für den Software-Zauberlehrling, R. Oldenbourg Verlag, München, 1990.
5. Baber, R.L., Praktische Anwendbarkeit mathematisch rigoroser Methoden zum Sicherstellen der Programmkorrektheit, Walter de Gruyter, Berlin, 1995.
6. Cook, S. “The P versus the NP Problem”, Clay Mathematics Institute, University of Toronto, www.claymath.org/prizeproblems/p_vs_np.pdf
7. Garey, M.R., Johnson, D.S. “Computers and Intractability: A Guide to the Theory of NP-Completeness” W.H. Feeman and Company, 1979.
8. Hopcroft, Motwani and Ullman, “Introduction to Automata”, Addison Wesley, 2nd edition.
9. Papadimitriou, Christos H., “Computational Complexity”, Addison-Wesley, 1994).