Fundamentals of Computer Algorithms

Leland Reyes

FUNDAMENTALS OF COMPUTER ALGORITHMS

FUNDAMENTALS OF COMPUTER ALGORITHMS

Leland Reyes

BIBLIOTEX
Digital Library

Fundamentals of Computer Algorithms
by Leland Reyes

Copyright© 2022 BIBLIOTEX

www.bibliotex.com

All rights reserved. No part of this book may be reproduced or used in any manner without the prior written permission of the copyright owner, except for the use brief quotations in a book review.

To request permissions, contact the publisher at info@bibliotex.com

Ebook ISBN: 9781984664211

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

1

Introduction to Algorithm

In mathematics and computer science, an algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Algorithms are used for calculation, data processing, and automated reasoning. Starting from an initial state and initial input (perhaps null), the instructions describe a computation that, when executed, will proceed through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input. A partial formalization of the concept began with attempts to solve the Entscheidungsproblem (the "decision problem") posed by David Hilbert in 1928. Subsequent formalizations were framed as attempts to define "effective calculability" or "effective method"; those formalizations included the Gödel–Herbrand–Kleene recursive functions of 1930, 1934 and 1935, Alonzo Church's lambda calculus of 1936, Emil Post's "Formulation 1" of 1936, and Alan Turing's Turing machines of 1936–7 and 1939.

WHY ALGORITHMS ARE NECESSARY?

While there is no generally accepted *formal* definition of "algorithm," an informal definition could be "a set of rules that precisely defines a sequence of operations." For some people, a programme is only an algorithm if it stops eventually; for others, a programme is only an algorithm if it stops before a given number of calculation steps.

A prototypical example of an algorithm is Euclid's algorithm to determine the maximum common divisor of two integers; an example (there are others) is described by the flow chart above and as an example in a later section. Boolos & Jeffrey (1974, 1999) offer an informal meaning of the word in the following quotation:

> No human being can write fast enough, or long enough, or small enough ("smaller and smaller without limit ...you'd be trying to write on molecules, on atoms, on electrons") to list all members of an enumerably infinite set by writing out their names, one after another, in some notation. But humans can do something equally useful, in the case of certain enumerably infinite sets: They can give explicit instructions for determining the *n*th member of the set, for arbitrary finite *n*. Such instructions are to be given quite explicitly, in a form in which they could

> be followed by a computing machine, or by a human who is capable of carrying out only very elementary operations on symbols

The term “enumerably infinite” means “countable using integers perhaps extending to infinity.” Thus Boolos and Jeffrey are saying that an algorithm implies instructions for a process that “creates” output integers from an *arbitrary* “input” integer or integers that, in theory, can be chosen from 0 to infinity. Thus an algorithm can be an algebraic equation such as $y = m + n$ — two arbitrary “input variables” m and n that produce an output y. But various authors’ attempts to define the notion indicate that the word implies much more than this, something on the order of (for the addition example):

> Precise instructions (in language understood by “the computer”) for a fast, efficient, “good” process that specifies the “moves” of “the computer” (machine or human, equipped with the necessary internally contained information and capabilities) to find, decode, and then process arbitrary input integers/symbols m and n, symbols + and = ... and “effectively” produce, in a “reasonable” time, output-integer y at a specified place and in a specified format.

The concept of *algorithm* is also used to define the notion of decidability. That notion is central for explaining how formal systems come into being starting from a small set of axioms and rules. In logic, the time that an algorithm requires to complete cannot be measured, as it is not apparently related with our customary physical dimension. From such uncertainties, that characterize ongoing work, stems the unavailability of a definition of *algorithm* that suits both concrete (in some sense) and abstract usage of the term.

FORMALIZATION

Algorithms are essential to the way computers process data. Many computer programmes contain algorithms that specify the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a Turing-complete system. Authors who assert this thesis include Minsky (1967), Savage (1987) and Gurevich (2000):

> Minsky: "But we will also maintain, with Turing . . . that any procedure which could "naturally" be called effective, can in fact be realized by a (simple) machine. Although this may seem extreme, the arguments . . . in its favour are hard to refute".
>
> Gurevich: "...Turing's informal argument in favour of his thesis justifies a stronger thesis: every algorithm can be simulated by a Turing machine ... according to Savage [1987], an algorithm is a computational process defined by a Turing machine".

Typically, when an algorithm is associated with processing information, data is read from an input source, written to an output device, and/or stored for further processing. Stored data is regarded as part of the internal state of the entity performing the algorithm. In practice, the state is stored in one or more data structures. For some such computational process, the algorithm must be rigorously defined: specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable). Because an algorithm is a precise list of precise steps, the order of computation will always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting "from the top" and going "down to the bottom", an idea that is described more formally by *flow of control.* So far, this discussion of the formalization of an algorithm has assumed the premises of imperative programming. This is the most common conception, and it attempts to describe a task in discrete, "mechanical" means. Unique to this conception of formalized algorithms is the assignment operation, setting the value of a variable. It derives from the intuition of "memory" as a scratchpad. There is an example below of such an assignment. For some alternate conceptions of what constitutes an algorithm see functional programming and logic programming.

TERMINATION

Some writers restrict the definition of *algorithm* to procedures that eventually finish. In such a category Kleene places the "*decision procedure* or *decision method* or *algorithm* for the question". Others, including Kleene, include procedures that could run forever without stopping; such a procedure has been called a "computational method" or "*calculation procedure* or *algorithm* (and hence a *calculation problem*) in relation to a general question which requires for an answer, not yes or no, but the exhibiting of some object". Minsky makes the pertinent observation, in regards to determining whether an algorithm will eventually terminate (from a particular starting state): But if the length of the process isn't known in advance, then "trying" it may not be decisive, because if the process does go on forever—then at no time will we ever be sure of the answer. As it happens, no other method can do any better, as was shown by Alan Turing with his celebrated result on the undecidability of the so-called halting problem. There is no algorithmic procedure for determining whether or not arbitrary algorithms terminate from given starting states. The analysis of algorithms for their likelihood of termination is called termination analysis.

See the examples of (im-) "proper" subtraction at partial function for more about what can happen when an algorithm fails for certain of its input numbers—e.g., (i) non-termination, (ii) production of "junk" (output in the wrong format to be considered a number) or no number(s) at all (halt ends the computation with no output), (iii) wrong number(s), or (iv) a combination of these. Kleene proposed that the production of "junk" or failure to produce a number is solved by having the algorithm detect these instances and produce e.g., an error message (he suggested "0"), or preferably, force the algorithm into an endless loop. Davis (1958) does this to his subtraction algorithm—he fixes his algorithm in a second example so that it is proper subtraction and it terminates. Along with the logical outcomes "true" and "false" Kleene (1952) also proposes the use of a third logical symbol "u" — undecided — thus an algorithm will always produce *something* when confronted with a "proposition". The problem of wrong answers must be solved with an independent "proof" of the algorithm e.g., using induction: We normally require auxiliary evidence for this [that the algorithm correctly defines a mu recursive function], e.g, in the form of an inductive proof that, for each argument value, the computation terminates with a unique value.

EXPRESSING ALGORITHMS

Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, programming languages or control tables (processed by interpreters). Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms. Pseudocode, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms. There is a wide variety of representations possible and one can express a given Turing machine programme as a sequence of machine tables as flowcharts, or as a form of rudimentary machine code or assembly code called "sets of quadruples". Sometimes it is helpful in the description of an algorithm to supplement small "flow charts" (state diagrams) with natural-language and/or arithmetic expressions written inside "block diagrams" to summarize what the "flow charts" are accomplishing. Representations of algorithms are generally classed into three accepted levels of Turing machine description:

- 1 High-level description:

  "...prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head."
- 2 Implementation description:

  "...prose used to define the way the Turing machine uses its head and the way that it stores data on its tape. At this level we do not give details of states or transition function."
- 3 Formal description:

  Most detailed, "lowest level", gives the Turing machine's "state table".

IMPLEMENTATION

Most algorithms are intended to be implemented as computer programmes. However, algorithms are also implemented by other means, such as in a biological neural network (for example, the human brain implementing arithmetic or an insect looking for food), in an electrical circuit, or in a mechanical device.

COMPUTER ALGORITHMS

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s), in order for the target machines to produce *output* from given *input* (perhaps null). "Elegant" (compact) programmes, "good" (fast) programmes : The notion of "simplicity and elegance" appears informally in Knuth and precisely in Chaitin:

> Knuth: ". . .we want *good* algorithms in some loosely defined aesthetic sense. One criterion . . . is the length of time taken to perform the algorithm . . .. Other criteria are adaptability of the algorithm to computers, its simplicity and elegance, etc"
>
> Chaitin: " . . . a programme is 'elegant,' by which I mean that it's the smallest possible programme for producing the output that it does"
>
> Chaitin prefaces his definition with: "I'll show you can't prove that a programme is 'elegant'" — such a proof would solve the Halting problem (ibid).

Algorithm versus function computable by an algorithm: For a given function multiple algorithms may exist. This will be true, even without expanding the available instruction set available to the programmer. Rogers observes that "It is . . . important to distinguish between the notion of *algorithm*, i.e. procedure and the notion of *function computable by algorithm*, i.e. mapping yielded by procedure. The same function may have several different algorithms". Unfortunately there may be a tradeoff between goodness (speed) and elegance (compactness) — an elegant programme may take more steps to complete a computation than one less elegant. An example of using Euclid's algorithm will be shown below. Computers (and computors), models of computation: A computer (or human "computor") is a restricted type of machine, a "discrete deterministic mechanical device" that blindly follows its instructions. Melzak's and Lambek's primitive models reduced this notion to four elements: (i) discrete, distinguishable *locations*, (ii) discrete, indistinguishable *counters* (iii) an agent, and (iv) a list of instructions that are *effective* relative to the capability of the agent.

Minsky describes a more congenial variation of Lambek's "abacus" model in his "Very Simple Bases for Computability". Minsky's machine proceeds sequentially through its five (or six depending on how one counts) instructions unless either a conditional IF - THEN GOTO or an unconditional GOTO changes programme flow out of sequence. Besides HALT, Minsky's machine includes three *assignment* (replacement, substitution) operations: ZERO, SUCCESSOR, and DECREMENT. Rarely will a programmer have to write "code" with such a limited instruction set. But Minsky shows (as do Melzak and Lambek) that his machine is Turing complete with only four general *types* of instructions: conditional GOTO, unconditional GOTO, assignment/replacement/ substitution, and HALT. Simulation of an algorithm: computer(computor) language: Knuth advises the reader that "the best way to learn an algorithm is to try it . . . immediately take pen and paper and work through an example". But what about a simulation or execution of the real thing? The programmer must translate the algorithm into a language that the simulator/computer/computor can *effectively* execute. Stone gives an example of this: when computing the roots of a quadratic equation the computor must know how to take a square root. If they don't then for the algorithm to be effective it must provide a set of rules for extracting a square root.

This means that the programmer must know a "language" that is effective relative to the target computing agent (computer/computor). But what model should be used for the simulation? Van Emde Boas observes "even if we base complexity theory on abstract instead of concrete machines, arbitrariness of the choice of a model remains. It is at this point that the notion of *simulation* enters". When speed is being measured, the instruction set matters. For example, the subprogramme in Euclid's algorithm to compute the remainder would execute much faster if the programmer had a "modulus" (division) instruction available rather than just subtraction (or worse: just Minsky's "decrement"). Structured programming, canonical structures: Per the Church-Turing thesis any algorithm can be computed by a model known to be Turing complete, and per Minsky's demonstrations Turing completeness requires only four instruction types—conditional GOTO, unconditional GOTO, assignment, HALT. Kemeny and Kurtz observe that while "undisciplined" use of unconditional GOTOs and conditional IF-THEN GOTOs can result in "spaghetti code" a programmer can write structured programmes using these instructions; on the other hand "it is also possible, and not too hard, to write badly structured programmes in a structured language".

Tausworthe augments the three Böhm-Jacopini canonical structures: SEQUENCE, IF-THEN-ELSE, and WHILE-DO, with two more: DO-WHILE and CASE. An additional benefit of a structured programme will be one that lends itself to proofs of correctness using mathematical induction. Canonical flowchart symbols: The graphical aide called a flowchart offers a way to describe and document an algorithm (and a computer programme of one). Like programme flow of a Minsky machine, a flowchart always starts at the top of a page and proceeds down. Its primary symbols are only 4: the directed arrow showing programme flow, the rectangle (SEQUENCE, GOTO), the diamond (IF-THEN-ELSE), and the dot (OR-tie). The Böhm-Jacopini canonical structures are made of these primitive shapes. Sub-structures can "nest" in rectangles but only if a single exit occurs from the superstructure. The symbols and their use to build the canonical structures are shown in the diagram.

EXAMPLES

SORTING EXAMPLE

One of the simplest algorithms is to find the largest number in an (unsorted) list of numbers. The solution necessarily requires looking at every number in the list, but only once at each. From this follows a simple algorithm, which can be stated in a high-level description English prose, as: High-level description:

1. Assume the first item is largest.
2. Look at each of the remaining items in the list and if it is larger than the largest item so far, make a note of it.
3. The last noted item is the largest in the list when the process is complete.

(Quasi-)formal description: Written in prose but much closer to the high-level language of a computer programme, the following is the more formal coding of the algorithm.

EUCLID'S ALGORITHM

Euclid's algorithm appears as Proposition II in Book VII ("Elementary Number Theory") of his *Elements*. Euclid poses the problem: "Given two numbers not prime to one another, to find their greatest common measure". He defines "A number [to be] a multitude composed of units": a counting number, a positive integer not including 0. And to "measure" is to place a shorter measuring length *s* successively (*q* times) along longer length *l* until the remaining portion *r* is less than the shorter length *s*. In modern words, remainder $r = l - q*s$, *q* being the quotient, or remainder *r* is the "modulus", the integer-fractional part left over after the division. For Euclid's method to succeed, the starting lengths must satisfy two requirements: (i) the lengths must not be 0, AND (ii) the subtraction must be "proper", a test must guarantee that the smaller of the two numbers is subtracted from the larger (alternately, the two can be equal so their subtraction yields 0). Euclid's original proof adds a third: the two lengths are not prime to one another. Euclid stipulated this so that he could construct a reductio ad absurdum proof that the two numbers' common measure is in fact the *greatest*. While Nicomachus' algorithm is the same as Euclid's, when the numbers are prime to one another it yields the number "1" for their common measure. So to be precise the following is really Nicomachus' algorithm.

COMPUTER (COMPUTOR) LANGUAGE FOR EUCLID'S ALGORITHM

Only a few instruction *types* are required to execute Euclid's algorithm—some logical tests (conditional GOTO), unconditional GOTO, assignment (replacement), and subtraction.

- A *location* is symbolized by upper case letter(s), e.g. S, A, etc.
- The varying quantity (number) in a location will be written in lower case letter(s) and (usually) associated with the location's name. For example, location L at the start might contain the number l = 3009.

How "Elegant" works: In place of an outer "Euclid loop", "Elegant" shifts back and forth between two "co-loops", an A > B loop that computes A ! A - B, and a B d" A loop that computes B ! B - A. This works because, when at last the minuend M is less than or equal to the subtrahend S (Difference = Minuend - Subtrahend), the minuend can become *s* (the new measuring length) and the subtrahend can become the new *r* (the length to be measured); in other words the "sense" of the subtraction reverses.

TESTING THE EUCLID ALGORITHMS

Does an algorithm do what its author wants it to do? A few test cases usually suffice to confirm core functionality. One source uses 3009 and 884. Knuth suggested 40902, 24140. Another interesting case is the two relatively-prime numbers 14157 and 5950. But exceptional cases must be identified and tested. Will "Inelegant" perform properly when R > S, S > R, R = S? Ditto for "Elegant": B > A, A > B, A

= B? (Yes to all). What happens when one number is zero, both numbers are zero? ("Inelegant" computes forever in all cases; "Elegant" computes forever when A = 0.) What happens if *negative* numbers are entered?

Fractional numbers? If the input numbers, i.e. the domain of the function computed by the algorithm/programme, is to include only positive integers including zero, then the failures at zero indicate that the algorithm (and the programme that instantiates it) is a partial function rather than a total function. A notable failure due to exceptions is the Ariane V rocket failure. Proof of programme correctness by use of mathematical induction: Knuth demonstrates the application of mathematical induction to an "extended" version of Euclid's algorithm, and he proposes "a general method applicable to proving the validity of any algorithm". Tausworthe proposes that a measure of the complexity of a programme be the length of its correctness proof.

MEASURING AND IMPROVING THE EUCLID ALGORITHMS

Elegance (compactness) versus goodness (speed) : With only 6 core instructions, "Elegant" is the clear winner compared to "Inelegant" at 13 instructions. However, "Inelegant" is *faster* (it arrives at HALT in fewer steps). Algorithm analysis indicates why this is the case: "Elegant" does *two* conditional tests in every subtraction loop, whereas "Inelegant" only does one. As the algorithm (usually) requires many loop-throughs, *on average* much time is wasted doing a "B = 0?" test that is needed only after the remainder is computed. Can the algorithms be improved?: Once the programmer judges a programme “fit” and “effective” — that is, it computes the function intended by its author—then the question becomes, can it be improved? The compactness of “Inelegant” can be improved by the elimination of 5 steps. But Chaitin proved that compacting an algorithm cannot be automated by a generalized algorithm; rather, it can only be done heuristically, i.e. by exhaustive search (examples to be found at Busy beaver), trial and error, cleverness, insight, application of inductive reasoning, etc. Observe that steps 4, 5 and 6 are repeated in steps 11, 12 and 13.

Comparison with “Elegant” provides a hint that these steps together with steps 2 and 3 can be eliminated. This reduces the number of core instructions from 13 to 8, which makes it “more elegant” than “Elegant” at 9 steps. The speed of “Elegant” can be improved by moving the B=0? test outside of the two subtraction loops. This change calls for the addition of 3 instructions (B=0?, A=0?, GOTO). Now “Elegant” computes the example-numbers faster; whether for any given A, B and R, S this is always the case would require a detailed analysis.

ALGORITHMIC ANALYSIS

It is frequently important to know how much of a particular resource (such as time or storage) is theoretically required for a given algorithm. Methods have been developed for the analysis of algorithms to obtain such quantitative answers (estimates); for example, the sorting algorithm above has a time requirement of O(n), using the big O notation with n as the length of the list. At all times the algorithm only needs to remember two values: the largest number found so far, and its current position in the input list. Therefore it is said to have a space requirement of *O(1)*, if the space required to store the input numbers is not counted, or O(*n*) if it is counted. Different algorithms may complete the same task with a different set of instructions in less or more time, space, or 'effort' than others. For example, a binary search algorithm will usually outperform a brute force sequential search when used for table lookups on sorted lists.

FORMAL VERSUS EMPIRICAL

The analysis and study of algorithms is a discipline of computer science, and is often practiced abstractly without the use of a specific programming language or implementation. In this sense, algorithm analysis resembles other mathematical disciplines in that it focuses on the underlying properties of the algorithm and not on the specifics of any particular implementation. Usually pseudocode is used for analysis as it is the simplest and most general representation. However, ultimately, most algorithms are usually implemented on particular hardware / software platforms and their algorithmic efficiency is eventually put to the test using real code. Empirical testing is useful because it may uncover unexpected interactions that affect performance. Benchmarks may be used to compare before/after potential improvements to an algorithm after programme optimization.

CLASSIFICATION

There are various ways to classify algorithms, each with its own merits.

BY IMPLEMENTATION

One way to classify algorithms is by implementation means.

- Recursion or iteration: A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other. For example, towers of Hanoi is well understood in recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.
- Logical: An algorithm may be viewed as controlled logical deduction. This notion may be expressed as: Algorithm = logic + control. The logic component expresses the axioms that may be used in the computation and the control component determines the way in which deduction is applied to the axioms. This is the basis for the logic programming paradigm. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant semantics: a change in the axioms has a well defined change in the algorithm.
- Serial or parallel or distributed: Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time.

Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms or distributed algorithms. Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilize multiple machines connected with a network. Parallel or distributed algorithms divide the problem into more symmetrical or asymmetrical subproblems and collect the results back together. The resource consumption in such algorithms is not only processor cycles on each processor but also the communication overhead between the processors. Sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems.

- Deterministic or non-deterministic: Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithms solve problems via guessing although typical guesses are made more accurate through the use of heuristics.
- Exact or approximate: While many algorithms reach an exact solution, approximation algorithms seek an approximation that is close to the true solution. Approximation may use either a deterministic or a random strategy. Such algorithms have practical value for many hard problems.

- Quantum algorithm: Quantum algorithm run on a realistic model of quantum computation. The term is usually used for those algorithms which seem inherently quantum, or use some essential feature of quantum computation such as quantum superposition or quantum entanglement.

BY DESIGN PARADIGM

Another way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include many different types of algorithms. Some commonly found paradigms include:

- Brute-force or exhaustive search. This is the naïve method of trying every possible solution to see which is best.
- Divide and conquer. A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in the conquer phase by merging the segments. A simpler variant of divide and conquer is called a decrease and conquer algorithm, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple

subproblems and so the conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is the binary search algorithm.

- Dynamic programming. When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems, and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, a quicker approach called *dynamic programming* avoids recomputing solutions that have already been computed. For example, Floyd–Warshall algorithm, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is that subproblems are more or less independent in divide and conquer, whereas subproblems overlap in dynamic programming. The difference between dynamic programming and straightforward recursion is in caching or memoization of recursive calls. When subproblems are independent and there is no repetition, memoization does not help; hence dynamic programming is not a solution for all complex problems. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

- The greedy method. A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the subproblems do not have to be known at each stage; instead a "greedy" choice can be made of what looks best for the moment. The greedy method extends the solution with the best possible decision (not all feasible decisions) at an algorithmic stage based on the current local optimum and the best decision (not all possible decisions) made in a previous stage. It is not exhaustive, and does not give an accurate answer to many problems. But when it works, it will be the fastest method. The most popular greedy algorithm is finding the minimal spanning tree as given by Huffman Tree, Kruskal, Prim, Sollin.
- Linear programming. When solving a problem using linear programming, specific inequalities involving the inputs are found and then an attempt is made to maximize (or minimize) some linear function of the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a linear programming way, and then be solved by a 'generic' algorithm such as the simplex algorithm. A more complex variant of linear programming is called integer programming, where the solution space is restricted to the integers.
- Reduction. This technique involves solving a difficult problem by transforming it into a better known problem for which we have (hopefully) asymptotically optimal algorithms. The goal is to find a reducing

algorithm whose complexity is not dominated by the resulting reduced algorithm's. For example, one selection algorithm for finding the median in an unsorted list involves first sorting the list (the expensive portion) and then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as *transform and conquer.*

- Search and enumeration. Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes search algorithms, branch and bound enumeration and backtracking.

1. Randomized algorithms are those that make some choices randomly (or pseudo-randomly); for some problems, it can in fact be proven that the fastest solutions must involve some randomness. There are two large classes of such algorithms:

(a) Monte Carlo algorithms return a correct answer with high-probability. E.g. RP is the subclass of these that run in polynomial time)

(b) Las Vegas algorithms always return the correct answer, but their running time is only probabilistically bound, e.g. ZPP.

2. In optimization problems, heuristic algorithms do not try to find an optimal solution, but an approximate solution where the time or resources are limited. They are not practical to find perfect solutions. An example of this would be local

search, tabu search, or simulated annealing algorithms, a class of heuristic probabilistic algorithms that vary the solution of a problem by a random amount. The name "simulated annealing" alludes to the metallurgic term meaning the heating and cooling of metal to achieve freedom from defects. The purpose of the random variance is to find close to globally optimal solutions rather than simply locally optimal ones, the idea being that the random element will be decreased as the algorithm settles down to a solution. Approximation algorithms are those heuristic algorithms that additionally provide some bounds on the error. Genetic algorithms attempt to find solutions to problems by mimicking biological evolutionary processes, with a cycle of random mutations yielding successive generations of "solutions". Thus, they emulate reproduction and "survival of the fittest". In genetic programming, this approach is extended to algorithms, by regarding the algorithm itself as a "solution" to a problem.

BY FIELD OF STUDY

Every field of science has its own problems and needs efficient algorithms. Related problems in one field are often studied together. Some example classes are search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, computational geometric algorithms, combinatorial algorithms, medical algorithms, machine learning, cryptography, data compression algorithms and parsing techniques. Fields tend to overlap with each other, and algorithm advances in one field may improve those of other, sometimes completely unrelated, fields. For example, dynamic programming was invented for optimization of resource consumption in industry, but is now used in solving a broad range of problems in many fields.

BY COMPLEXITY

Algorithms can be classified by the amount of time they need to complete compared to their input size. There is a wide variety: some algorithms complete in linear time relative to input size, some do so in an exponential amount of time or even worse, and some never halt. Additionally, some problems may have multiple algorithms of differing complexity, while other problems might have no algorithms or no known efficient algorithms. There are also mappings from some problems to other problems. Owing to this, it was found to be more suitable to classify the problems themselves instead of the algorithms into equivalence classes based on the complexity of the best possible algorithms for them. Burgin (2005, p. 24) uses a generalized definition of algorithms that relaxes the common requirement that the output of the algorithm that computes a function must be determined after a finite number of steps. He defines a super-recursive class of algorithms as "a class of algorithms in which it is possible to compute functions not computable by any Turing machine" (Burgin 2005, p. 107). This is closely related to the study of methods of hypercomputation.

CONTINUOUS ALGORITHMS

The adjective "continuous" when applied to the word "algorithm" can mean:

1. An algorithm operating on data that represents continuous quantities, even though this data is represented by discrete approximations – such algorithms are studied in numerical analysis; or
2. An algorithm in the form of a differential equation that operates continuously on the data, running on an analog computer.

LEGAL ISSUES

Algorithms, by themselves, are not usually patentable. In the United States, a claim consisting solely of simple manipulations of abstract concepts, numbers, or signals does not constitute "processes" (USPTO 2006), and hence algorithms are not patentable (as in Gottschalk v. Benson). However, practical applications of algorithms are sometimes patentable. For example, in Diamond v. Diehr, the application of a simple feedback algorithm to aid in the curing of synthetic rubber was deemed patentable. The patenting of software is highly controversial, and there are highly criticized patents involving algorithms, especially data compression algorithms, such as Unisys' LZW patent. Additionally, some cryptographic algorithms have export restrictions.

ETYMOLOGY

The word *"Algorithm"* or *"Algorism"* in some other writing versions, comes from the name Al-Khwarizmi (c. 780-850), a Persian mathematician, astronomer, geographer and a scholar in the House of Wisdom in Baghdad, whose name means *"the native of Kharazm"*, a city that was part of the Greater Iran during his era and now is in modern day

Uzbekistan He wrote a treatise in Arabic language in the 9th century, which was translated into Latin in the 12th century under the title *Algoritmi de numero Indorum*. This title means "Algoritmi on the numbers of the Indians", where "Algoritmi" was the translator's Latinization of Al-Khwarizmi's name. Al-Khwarizmi was the most widely read mathematician in Europe in the late Middle Ages, primarily through his other book, the Algebra. In late medieval Latin, *algorismus*, the corruption of his name, simply meant the "decimal number system" that is still the meaning of modern English algorism. In 17th century French the word's form, but not its meaning, changed to *algorithme*. English adopted the French very soon afterwards, but it wasn't until the late 19th century that "Algorithm" took on the meaning that it has in modern English.

ALGORITHM CHARACTERIZATIONS

The word algorithm does not have a generally accepted definition. Researchers are actively working in formalizing this term. This study will present some of the "characterizations" of the notion of "algorithm" in more detail.

THE PROBLEM OF DEFINITION

There is no generally accepted definition of algorithm. Over the last 200 years the definition has become more complicated and detailed as researchers have tried to pin down the term. Indeed there may be more than one type of "algorithm". But most agree that algorithm has something to do with defining generalized processes for the creation of “output” integers from other “input” integers — “input parameters” arbitrary and infinite in extent, or limited in extent but still variable—by the manipulation of distinguishable symbols (counting numbers) with finite collections of rules that a person can perform with paper and pencil. The most common number-manipulation schemes—both in formal mathematics and in routine life—are: (1) the recursive functions calculated by a person with paper and pencil, and (2) the Turing machine or its Turing equivalents—the primitive register machine or “counter machine” model, the Random Access Machine model (RAM), the Random access stored programme machine model (RASP) and its functional equivalent “the computer”.

The reader is probably unfamiliar with the notion of a “recursive function”. But when we are doing “arithmetic” we are really calculating by the use of “recursive functions” in the shorthand algorithms we learned in grade-school, for example, adding and subtracting. The proofs that every “recursive function” we can *calculate by hand* we can *compute by machine* and vice versa—note the usage of the words *calculate* versus *compute* — is remarkable. But this equivalence together with the *thesis* (hypothesis, unproven assertion) that this includes *every* calculation/computation indicates why so much emphasis has been placed upon the use of Turing-equivalent machines in the definition of specific algorithms, and why the definition of “algorithm” itself often refers back to “the Turing machine”. This is discussed in more detail under Stephen Kleene’s characterization. The following are summaries of the more famous characterizations (Kleene, Markov, Knuth) together with those that introduce novel elements—elements that further expand the definition or contribute to a more precise definition.

CHOMSKY HIERARCHY

There is more consensus on the "characterization" of the notion of "simple algorithm". All algorithms need to be specified in a formal language, and the "simplicity notion" arises from the simplicity of the language. The Chomsky (1956) hierarchy is a containment hierarchy of classes of formal grammars that generate formal languages. It is used for classifying of programming languages and abstract machines. From the *Chomsky hierarchy* perspective, if the algorithm can be specified on a simpler language (than unrestricted), it can be characterized by this kind of language, else it is a typical "unrestricted algorithm". Examples: a "general purpose" macro language, like M4 is unrestricted (Turing complete), but the C preprocessor macro language is not, so any algorithm expressed in *C preprocessor* is a "simple algorithm".

CHARACTERIZATIONS OF THE NOTION OF "ALGORITHM"

1881 John Venn's negative reaction to W. Stanley Jevons's Logical Machine of 1870 In early 1870 W. Stanley Jevons presented a "Logical Machine" (Jevons 1880:200) for analyzing a syllogism or other logical form e.g. an argument reduced to a Boolean equation. By means of what Couturat (1914) called a "sort of *logical piano* [,] ... the equalities which represent the premises ... are "played" on a keyboard like that of a typewriter. ... When all the premises have been "played", the panel shows only those constituents whose sum is equal to 1, that is, ... its logical whole. This mechanical method has the advantage over VENN's geometrical method..." (Couturat 1914:75). For his part John Venn, a logician contemporary to Jevons, was less than thrilled, opining that "it does not seem to me that any contrivances at present known *or likely to be discovered* really deserve the name of logical machines" (italics added, Venn 1881:120). But of historical use to the developing notion of "algorithm" is his explanation for his negative reaction with respect to a machine that "may subserve a really valuable purpose by enabling us to avoid otherwise inevitable labor":

(1) "There is, first, the statement of our data in accurate logical language",

(2) "Then secondly, we have to throw these statements into a form fit for the engine to work with — in this case the reduction of each proposition to its elementary denials",

(3) "Thirdly, there is the combination or further treatment of our premises after such reduction,"

(4) "Finally, the results have to be interpreted or read off. This last generally gives rise to much opening for skill and sagacity."

He concludes that "I cannot see that any machine can hope to help us except in the third of these steps; so that it seems very doubtful whether any thing of this sort really deserves the name of a logical engine."(Venn 1881:119-121).

1943, 1952 STEPHEN KLEENE'S CHARACTERIZATION

This section is longer and more detailed than the others because of its importance to the topic: Kleene was the first to propose that *all* calculations/computations—of *every* sort, the *totality* of—can *equivalently* be (i) *calculated* by use of five "primitive recursive operators" plus one special operator called the mu-operator, or be (ii) *computed* by the actions of a Turing machine or an equivalent model. Furthermore he opined that either of these would stand as a definition of algorithm. A reader first confronting the words that follow may well be confused, so a brief explanation is in order. *Calculation* means done by hand, *computation* means done by Turing machine (or equivalent). (Sometimes an author slips and interchanges the words). A "function" can be thought of as an "input-output box" into which a person puts natural numbers called "arguments" or "parameters" (but only the counting numbers including 0—the positive integers) and gets out a single positive integer (including 0) (conventionally called "the answer"). Think of the "function-box" as a little man either calculating by hand using "general recursion" or computing by Turing machine (or an equivalent machine). "Effectively calculable/computable" is more generic and means "calculable/computable by *some* procedure, method, technique ... whatever...". "General recursive" was Kleene's way of writing what today is called just "recursion"; however, "primitive recursion" — calculation by use of the five recursive operators—is a lesser form of recursion that lacks access to the sixth, additional, mu-operator that is needed only in rare instances. Thus most of life goes on requiring only the "primitive recursive functions."

1943 "THESIS I", 1952 "CHURCH'S THESIS"

In 1943 Kleene proposed what has come to be known as Church's thesis:

> "*Thesis I.* Every effectively calculable function (effectively decidable predicate) is general recursive" (First stated by Kleene in 1943 (reprinted page 274 in Davis, ed. *The Undecidable*; appears also verbatim in Kleene (1952) p.300)

In a nutshell: to calculate *any* function the only operations a person needs (technically, formally) are the 6 primitive operators of "general" recursion (nowadays called the operators of the mu recursive functions). Kleene's first statement of this was under the section title "12. Algorithmic theories". He would later amplify it in his text (1952) as follows:

> "Thesis I and its converse provide the exact definition of the notion of a calculation (decision) procedure or algorithm, for the case of a function (predicate) of natural numbers" (p. 301, boldface added for emphasis)

(His use of the word "decision" and "predicate" extends the notion of calculability to the more general manipulation of symbols such as occurs in mathematical "proofs".) This is not as daunting as it may sound — "general" recursion is just a way of making our everyday arithmetic operations from the five "operators" of the primitive recursive functions together with the additional mu-operator as needed. Indeed, Kleene gives 13 examples of primitive recursive functions and Boolos-Burgess-Jeffrey add some more, most of which will be familiar to the reader—e.g. addition, subtraction, multiplication and division, exponentiation, the CASE function, concatenation, etc, etc; for a list see Some common primitive recursive functions. Why general-recursive functions rather than primitive-recursive functions?

Kleene et al. (cf §55 General recursive functions p. 270 in Kleene 1952) had to add a sixth recursion operator called the minimization-operator (written as ì-operator or mu-operator) because Ackermann (1925) produced a hugely-growing function—the Ackermann function — and Rózsa Péter (1935) produced a general method of creating recursive functions using Cantor's diagonal argument, neither of which could be described by the 5 primitive-recursive-function operators. With respect to the Ackermann function:

> "...in a certain sense, the length of the computation [*sic*] algorithm of a recursive function which is not also primitive recursive grows faster with the arguments than the value of any primitive recursive function" (Kleene (1935) reprinted p. 246 in *The Undecidable*, plus footnote 13 with regards to the need for an additional operator, boldface added).

But the need for the mu-operator is a rarity. As indicated above by Kleene's list of common calculations, a person goes about their life happily computing primitive recursive functions without fear of encountering the monster numbers created by Ackermann's function (e.g. super-exponentiation).

1952 "TURING'S THESIS"

Turing's Thesis hypothesizes the computability of "all computable functions" by the Turing machine model and its equivalents. To do this in an effective manner, Kleene extended the notion of "computable" by casting the net wider—by allowing into the notion of "functions" both "total functions" and "partial functions". A *total function* is one that is defined *for all natural numbers* (positive integers including 0). A partial function is defined for *some* natural numbers but not all—the specification of "some" has to come "up front". Thus the inclusion of "partial function" extends the notion of function to "less-perfect" functions. Total- and partial-functions may either be calculated by hand or computed by machine.

EXAMPLES:

"Functions": include "common subtraction m-n" and "addition m+n" "Partial function": "Common subtraction" m-n is undefined when only natural numbers (positive integers and zero) are allowed as input — e.g. 6-7 is undefined Total function: "Addition" m+n is defined for all positive integers and zero. We now observe Kleene's definition of "computable" in a formal sense: Definition: "A partial function ö is *computable*, if there is a machine M which computes it" (Kleene (1952) p. 360) "Definition 2.5. An n-ary function $f(x_1,\ldots x_n)$ is partially computable if there exists a Turing machine Z such that

$$f(x_1,\ldots x_n) = Ø_Z^{(n)}(x_1,\ldots x_n)$$

In this case we say that [machine] Z computes f. If, in addition, $f(x_1,\ldots x_n)$ is a total function, then it is called computable" (Davis (1958) p. 10). Thus we have arrived at *Turing's Thesis*:

> "Every function which would naturally be regarded as computable is computable ... by one of his machines..." (Kleene (1952) p.376)

Although Kleene did not give examples of "computable functions" others have. For example, Davis (1958) gives Turing tables for the Constant, Successor and Identity functions, three of the five operators of the primitive recursive functions: Computable by Turing machine:

Addition (also is the Constant function if one operand is 0)

Increment (Successor function)

Common subtraction (defined only if x e" y). Thus "x - y" is an example of a partially computable function. Proper subtraction x4%y (as defined above) The identity function: for each i, a function $U_Z^n = Ø_Z^n(x_1,\dots x_n)$ exists that plucks x_i out of the set of arguments $(x_1,\dots x_n)$

MULTIPLICATION

Boolos-Burgess-Jeffrey (2002) give the following as prose descriptions of Turing machines for:

Doubling: 2*p

Parity

Addition

MULTIPLICATION

With regards to the counter machine, an abstract machine model equivalent to the Turing machine:

Examples Computable by Abacus machine (cf Boolos-Burgess-Jeffrey (2002))

Addition

Multiplication

Exponention: (a flow-chart/block diagram description of the algorithm)

Demonstrations of computability by abacus machine (Boolos-Burgess-Jeffrey (2002)) and by counter machine (Minsky 1967): The six recursive function operators:

1. Zero function
2. Successor function
3. Identity function
4. Composition function
5. Primitive recursion (induction)
6. Minimization

The fact that the abacus/counter machine models can simulate the recursive functions provides the proof that: If a function is "machine computable" then it is "hand-calculable by partial recursion". Kleene's Theorem XXIX :

"*Theorem XXIX: "Every computable partial function ö is partial recursive...*" (italics in original, p. 374).

The converse appears as his Theorem XXVIII. Together these form the proof of their equivalence, Kleene's Theorem XXX.

1952 CHURCH-TURING THESIS

With his Theorem XXX Kleene proves the *equivalence* of the two "Theses" — the Church Thesis and the Turing Thesis. (Kleene can only hypothesize (conjecture) the truth of both thesis — *these he has not proven*):

> *THEOREM XXX: The following classes of partial functions ... have the same members: (a) the partial recursive functions, (b) the computable functions ...*"(p. 376)
>
> Definition of "partial recursive function": "A partial function ö is partial recursive in [the partial functions] $ø_1$, ... $ø_n$ if there is a system of equations E which defines ö recursively from [partial functions] $ø_1$, ... $ø_n$" (p. 326)

Thus by Kleene's Theorem XXX: either method of making numbers from input-numbers—recursive functions calculated by hand or computated by Turing-machine or equivalent—results in an "*effectively calculable/computable* function".

If we accept the hypothesis that *every* calculation/ computation can be done by either method equivalently we have accepted both Kleene's Theorem XXX (the equivalence) and the Church-Turing Thesis (the hypothesis of "every").

A BOTE OF DISSENT: "THERE'S MORE TO ALGORITHM..." BLASS AND GUREVICH (2003)

The notion of separating out Church's and Turing's theses from the "Church-Turing thesis" appears not only in Kleene (1952) but in Blass-Gurevich (2003) as well. But there while there are agreements, there are disagreements too:

> "...we disagree with Kleene that the notion of algorithm is that well understood. In fact the notion of algorithm is richer these days than it was in Turing's days. And there are algorithms, of modern

> and classical varieties, not covered directly by Turing's analysis, for example, algorithms that interact with their environments, algorithms whose inputs are abstract structures, and geometric or, more generally, non-discrete algorithms" (Blass-Gurevich (2003) p. 8, boldface added)

1954 A.A. MARKOV'S CHARACTERIZATION

A. A. Markov (1954) provided the following definition of algorithm:

> "1. In mathematics, "algorithm" is commonly understood to be an exact prescription, defining a computational process, leading from various initial data to the desired result...."

"The following three features are characteristic of algorithms and determine their role in mathematics:

"a) the precision of the prescription, leaving no place to arbitrariness, and its universal comprehensibility — the definiteness of the algorithm;

"b) the possibility of starting out with initial data, which may vary within given limits — the generality of the algorithm;

"c) the orientation of the algorithm toward obtaining some desired result, which is indeed obtained in the end with proper initial data — the conclusiveness of the algorithm." (p.1)

He admitted that this definition "does not pretend to mathematical precision" (p. 1). His 1954 monograph was his attempt to define algorithm more accurately; he saw his resulting definition—his "normal" algorithm—as "equivalent to the concept of a recursive function" (p. 3). His definition included four major components (Chapter II.3 pp. 63ff):

"1. Separate elementary steps, each of which will be performed according to one of [the substitution] rules... [rules given at the outset]

"2. ... steps of local nature ... [Thus the algorithm won't change more than a certain number of symbols to the left or right of the observed word/symbol]

"3. Rules for the substitution formulas ... [he called the list of these "the scheme" of the algorithm]

"4. ...a means to distinguish a "concluding substitution" [i.e. a distinguishable "terminal/final" state or states]

In his Introduction Markov observed that "the entire significance for mathematics" of efforts to define algorithm more precisely would be "in connection with the problem of a constructive foundation for mathematics" (p. 2). Ian Stewart (cf Encyclopedia Britannica) shares a similar belief: "...constructive analysis is very much in the same algorithmic spirit as computer science...". For more see constructive mathematics and Intuitionism. Distinguishability and Locality: Both notions first appeared with Turing (1936–1937) —

> "The new observed squares must be immediately recognizable by the computer [*sic*: a computer was a person in 1936]. I think it reasonable to suppose that they can only be squares whose distance from the closest of the immediately observed squares does not exceed a certain fixed amount.

> Let us stay that each of the new observed squares is within L squares of one of the previously observed squares." (Turing (1936) p. 136 in Davis ed. *Undecidable*)

Locality appears prominently in the work of Gurevich and Gandy (1980) (whom Gurevich cites). Gandy's "Fourth Principle for Mechanisms" is "The Principle of Local Causality":

> "We now come to the most important of our principles. In Turing's analysis the requirement that the action depend only on a bounded portion of the record was based on a human limitiation. We replace this by a physical limitation which we call the *principle of local causation.* Its justification lies in the finite velocity of propagation of effects and signals: contemporary physics rejects the possibility of instantaneous action at a distance." (Gandy (1980) p. 135 in J. Barwise et al.)

1936, 1963, 1964 GODEL'S CHARACTERIZATION

1936: A rather famous quote from Kurt Gödel appears in a "Remark added in proof [of the original German publication] in his paper "On the Length of Proofs" translated by Martin Davis appearing on pp. 82–83 of *The Undecidable*. A number of authors—Kleene, Gurevich, Gandy etc. — have quoted the following:

> "Thus, the concept of "computable" is in a certain definite sense "absolute," while practically all other familiar metamathematical concepts (e.g. provable,

> definable, etc.) depend quite essentially on the system with respect to which they are defined." (p. 83)

1963: In a "Note" dated 28 August 1963 added to his famous paper *On Formally Undecidable Propostions* (1931) Gödel states (in a footnote) his belief that "formal systems" have "the characteristic property that reasoning in them, in principle, can be completely replaced by mechanical devices" (p. 616 in van Heijenoort). ". . . due to "A. M. Turing's work a precise and unquestionaly adequate definition of the general notion of formal system can now be given [and] a completely general version of Theorems VI and XI is now possible." (p. 616). In a 1964 note to another work he expresses the same opinion more strongly and in more detail.

1964: In a Postscriptum, dated 1964, to a paper presented to the Institute for Advanced Study in spring 1934, Gödel amplified his conviction that "formal systems" are those that can be mechanized:

> "In consequence of later advances, in particular of the fact that, due to A. M. Turing's work, a precise and unquestionably adequate definition of the general concept of formal system can now be given . . . Turing's work gives an analysis of the concept of "mechanical procedure" (alias "algorithm" or "computational procedure" or "finite combinatorial procedure"). This concept is shown to be equivalent with that of a "Turing machine".* A formal system can simply be defined to be any mechanical procedure for producing formulas,

> called provable formulas . . . ." (p. 72 in Martin Davis ed. *The Undecidable*: "Postscriptum" to "On Undecidable Propositions of Formal Mathematical Systems" appearing on p. 39, loc. cit.)

The * indicates a footnote in which Gödel cites the papers by Allan Turing (1937) and Emil Post (1936) and then goes on to make the following intriguing statement:

> "As for previous equivalent definitions of computability, which however, are much less suitable for our purpose, see Alonzo Church, Am. J. Math., vol. 58 (1936) [appearing in *The Undecidable* pp. 100-102]).

Church's definitions encompass so-called "recursion" and the "lambda calculus" (i.e. the ë-definable functions). His footnote 18 says that he discussed the relationship of "effective calculatibility" and "recursiveness" with Gödel but that he independently questioned "effectively calculability" and "ë-definability":

> "We now define the notion . . . of an effectively calculable function of positive integers by identifying it with the notion of a recursive function of positive integers[18] (or of a ë-definable function of positive integers.
>
> "It has already been pointed out that, for every function of positive integers which is effectively calculable in the sense just defined, there exists an algorithm for the calculation of its value.
>
> "Conversely it is true . . ." (p. 100, The Undecidable).

It would appear from this, and the following, that far as Gödel was concerned, the Turing machine was sufficient and the lambda calculus was "much less suitable."

He goes on to make the point that, with regards to limitations on human reason, the jury is still out:

> ("Note that the question of whether there exist finite non-mechanical procedures not equivalent with any algorithm, has nothing whatsoever to do with the adequacy of the definition of "formal system" and of "mechanical procedure.") (p. 72, loc. cit.)
>
> "(For theories and procedures in the more general sense indicated in footnote the situation may be different. Note that the results mentioned in the postcript do not establish any bounds for the powers of human reason, but rather for the potentialities of pure formalism in mathematics.) (p. 73 loc. cit.)

1967 MINSKY'S CHARACTERIZATION

Minsky (1967) baldly asserts that "an algorithm is "an effective procedure" and declines to use the word "algorithm" further in his text; in fact his index makes it clear what he feels about "Algorithm, *synonym* for Effective procedure"(p. 311):

> "We will use the latter term [an *effective procedure*] in the sequel. The terms are roughly synonymous, but there are a number of shades of meaning used in different contexts, especially for 'algorithm'" (italics in original, p. 105)

Other writers use the word “effective procedure”.

This leads one to wonder: What is Minsky’s notion of “an effective procedure”? He starts off with:

> “...a set of rules which tell us, from moment to moment, precisely how to behave” (p. 106)

But he recognizes that this is subject to a criticism:

> “... the criticism that the interpretation of the rules is left to depend on some person or agent” (p. 106)

His refinement? To “specify, along with the statement of the rules, *the details of the mechanism that is to interpret them*”. To avoid the “cumbersome” process of “having to do this over again for each individual procedure” he hopes to identify a “reasonably *uniform* family of rule-obeying mechanisms”. His “formulation”:

> “(1) a *language* in which sets of behavioural rules are to be expressed, and “(2) a *single* machine which can interpret statements in the language and thus carry out the steps of each specified process.” (italics in original, all quotes this para. p. 107)

In the end, though, he still worries that “there remains a subjective aspect to the matter. Different people may not agree on whether a certain procedure should be called effective” (p. 107). But Minsky is undeterred. He immediately introduces “Turing’s Analysis of Computation Process” (his chapter 5.2). He quotes what he calls “Turing’s *thesis*”

> “Any process which could naturally be called an effective procedure can be realized by a Turing

> machine" (p. 108. (Minsky comments that in a more general form this is called "*Church's thesis*").

After an analysis of "Turing's Argument" (his chapter 5.3) he observes that "equivalence of many intuitive formulations" of Turing, Church, Kleene, Post, and Smullyan "…leads us to suppose that there is really here an 'objective' or 'absolute' notion. As Rogers [1959] put it:

> "In this sense, the notion of effectively computable function is one of the few 'absolute' concepts produced by modern work in the foundations of mathematics'" (Minsky p. 111 quoting Rogers, Hartley Jr (1959) *The present theory of Turing machine computability*, J. SIAM 7, 114-130.)

1967 ROGERS' CHARACTERIZATION

In his 1967 *Theory of Recursive Functions and Effective Computability* Hartley Rogers' characterizes "algorithm" roughly as "a clerical (i.e., deterministic, bookkeeping) procedure . . . applied to . . . symbolic *inputs* and which will eventually yield, for each such input, a corresponding symbolic *output*"(p. 1). He then goes on to describe the notion "in approximate and intuitive terms" as having 10 "features", 5 of which he asserts that "virtually all mathematicians would agree [to]" (p. 2). The remaining 5 he asserts "are less obvious than *1 to *5 and about which we might find less general agreement" (p. 3). The 5 "obvious" are:

- 1 An algorithm is a set of instructions of finite size,
- 2 There is a capable computing agent,

- 3 "There are facilities for making, storing, and retrieving steps in a computation"
- 4 Given #1 and #2 the agent computes in "discrete stepwise fashion" without use of continuous methods or analogue devices",
- 5 The computing agent carries the computation forward "without resort to random methods or devices, e.g., dice" (in a footnote Rogers wonders if #4 and #5 are really the same)

The remaining 5 that he opens to debate, are:

- 6 No fixed bound on the size of the inputs,
- 7 No fixed bound on the size of the set of instructions,
- 8 No fixed bound on the amount of memory storage available,
- 9 A fixed finite bound on the capacity or ability of the computing agent (Rogers illustrates with example simple mechanisms similar to a Post-Turing machine or a counter machine),
- 10 A bound on the length of the computation — "should we have some idea, 'ahead of time', how long the computationwill take?" (p. 5). Rogers requires "only that a computation terminate after *some* finite number of steps; we do not insist on an a priori ability to estimate this number." (p. 5).

1968, 1973 KNUTH'S CHARACTERIZATION

Knuth (1968, 1973) has given a list of five properties that are widely accepted as requirements for an algorithm:

1. Finiteness: “An algorithm must always terminate after a finite number of steps ... a *very* finite number, a reasonable number”
2. Definiteness: “Each step of an algorithm must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case”
3. Input: “...quantities which are given to it initially before the algorithm begins. These inputs are taken from specified sets of objects”
4. Output: “...quantities which have a specified relation to the inputs”
5. Effectiveness: “... all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time by a man using paper and pencil”

Knuth offers as an example the Euclidean algorithm for determining the greatest common divisor of two natural numbers (cf. Knuth Vol. 1 p. 2). Knuth admits that, while his description of an algorithm may be intuitively clear, it lacks formal rigor, since it is not exactly clear what “precisely defined” means, or “rigorously and unambiguously specified” means, or “sufficiently basic”, and so forth. He makes an effort in this direction in his first volume where he defines *in detail* what he calls the “machine language” for his “mythical MIX...the world’s first polyunsaturated computer” (pp. 120ff). Many of the algorithms in his books are written in the MIX language. He also uses tree diagrams, flow diagrams and state diagrams.

> "Goodness" of an algorithm, "best" algorithms: Knuth states that "In practice, we not only want algorithms, we want *good* algorithms...." He suggests that some criteria of an algorithm's goodness are the number of steps to perform the algorithm, its "adaptability to computers, its simplicity and elegance, etc." Given a number of algorithms to perform the same computation, which one is "best"? He calls this sort of inquiry "algorithmic analysis: given an algorithm, to determine its performance characteristcis" (all quotes this paragraph: Knuth Vol. 1 p. 7)

1972 STONE'S CHARACTERIZATION

Stone (1972) and Knuth (1968, 1973) were professors at Stanford University at the same time so it is not surprising if there are similarities in their definitions (boldface added for emphasis):

> "To summarize ... we define an algorithm to be a set of rules that precisely defines a sequence of operations such that each rule is effective and definite and such that the sequence terminates in a finite time." (boldface added, p. 8)

Stone is noteworthy because of his detailed discussion of what constitutes an "effective" rule – his robot, or person-acting-as-robot, must have some information and abilities within them, and if not the information and the ability must be provided in "the algorithm":

> "For people to follow the rules of an algorithm, the rules must be formulated so that they can be

> followed in a robot-like manner, that is, without the need for thought... however, if the instructions [to solve the quadratic equation, his example] are to be obeyed by someone who knows how to perform arithmetic operations but does not know how to extract a square root, then we must also provide a set of rules for extracting a square root in order to satisfy the definition of algorithm" (p. 4-5)

Furthermore "...not all instructions are acceptable, because they may require the robot to have abilities beyond those that we consider reasonable." He gives the example of a robot confronted with the question is "Henry VIII a King of England?" and to print 1 if yes and 0 if no, but the robot has not been previously provided with this information. And worse, if the robot is asked if Aristotle was a King of England and the robot only had been provided with five names, it would not know how to answer. Thus:

> "an intuitive definition of an acceptable sequence of instructions is one in which each instruction is precisely defined so that the robot is guaranteed to be able to obey it" (p. 6)

After providing us with his definition, Stone introduces the Turing machine model and states that the set of five-tuples that are the machine's instructions are "an algorithm ... known as a Turing machine program" (p. 9). Immediately thereafter he goes on say that a "*computation* of a Turing machine is *described* by stating:

"1. The tape alphabet

"2. The form in which the [input] parameters are presented on the tape

"3. The initial state of the Turing machine

"4. The form in which answers [output] will be represented on the tape when the Turing machine halts

"5. The machine programme" (italics added, p. 10)

This precise prescription of what is required for "a computation" is in the spirit of what will follow in the work of Blass and Gurevich.

1995 SOARE'S CHARACTERIZATION

"A *computation* is a process whereby we proceed from initially given objects, called *inputs*, according to a fixed set of rules, called a *programme, procedure,* or *algorithm*, through a series of *steps* and arrive at the end of these steps with a final result, called the *output*. The algorithm, as a set of rules proceeding from inputs to output, must be precise and definite with each successive step clearly determined. The concept of *computability* concerns those objects which may be specified in principle by computations . . ."

2000 BERLINSKI'S CHARACTERIZATION

While a student at Princeton in the mid-1960s, David Berlinski was a student of Alonzo Church (cf p. 160). His year-2000 book *The Advent of the Algorithm: The 300-year Journey from an Idea to the Computer* contains the following definition of algorithm:

"*In the logician's voice:*

"*an algorithm is*

a finite procedure,

written in a fixed symbolic vocabulary,

governed by precise instructions,
moving in discrete steps, 1, 2, 3, . . .,
whose execution requires no insight, cleverness,
intuition, intelligence, or perspicuity,
and that sooner or later comes to an end.'" (boldface and italics in the original, p. xviii)

2000, 2002 GUREVICH'S CHARACTERIZATION

A careful reading of Gurevich 2000 leads one to conclude (infer?) that he believes that "an algorithm" is actually "a Turing machine" or "a pointer machine" doing a computation. An "algorithm" is not just the symbol-table that guides the behaviour of the machine, nor is it just one instance of a machine doing a computation given a particular set of input parameters, nor is it a suitably-programmed machine with the power off; rather *an algorithm is the machine actually doing any computation of which it is capable*. Gurevich does not come right out and say this, so as worded above this conclusion (inference?) is certainly open to debate:

> " . . . every algorithm can be simulated by a Turing machine . . . a programme can be simulated and therefore given a precise meaning by a Turing machine." (p. 1)

> " It is often thought that the problem of formalizing the notion of sequential algorithm was solved by Church [1936] and Turing [1936]. For example, according to Savage [1987], an algorithm is a computational *process* defined by a Turing machine. Church and Turing did not solve the

> problem of formalizing the notion of sequential algorithm. Instead they gave (different but equivalent) formalizations of the notion of computable function, and there is more to an algorithm than the function it computes. (italics added p. 3)
>
> "Of course, the notions of algorithm and computable function are intimately related: by definition, a computable function is a function computable by an algorithm. . . . (p. 4)

In Blass and Gurevich 2002 the authors invoke a dialog between "Quisani" ("Q") and "Authors" (A), using Yiannis Moshovakis as a foil, where they come right out and flatly state:

> "A: To localize the disagreement, let's first mention two points of agreement. First, there are some things that are obviously algorithms by anyone's definition — Turing machines, sequential-time ASMs [Abstract State Machines], and the like. . . .Second, at the other extreme are specifications that would not be regarded as algorithms under anyone's definition, since they give no indication of how to compute anything . . . The issue is how detailed the information has to be in order to count as an algorithm. . . . Moshovakis allows some things that we would call only declarative specifications, and he would probably use the word "implementation" for things that we call algorithms." (paragraphs joined for ease of readability, 2002:22)

This use of the word "implementation" cuts straight to the heart of the question. Early in the paper, Q states his reading of Moshovakis:

> "...[H]e would probably think that your practical work [Gurevich works for Microsoft] forces you to think of implementations more than of algorithms. He is quite willing to identify implementations with machines, but he says that algorithms are something more general. What it boils down to is that you say an algorithm is a machine and Moschovakis says it is not." (2002:3)

But the authors waffle here, saying "[L]et's stick to "algorithm" and "machine", and the reader is left, again, confused. We have to wait until Dershowitz and Gurevich 2007 to get the following footnote comment:

> " . . . Nevertheless, if one accepts Moshovakis's point of view, then it is the "implementation" of algorithms that we have set out to characterize."(cf Footnote 9 2007:6)

2003 BLASS AND GUREVICH'S CHARACTERIZATION

Blass and Gurevich describe their work as evolved from consideration of Turing machines and pointer machines, specifically Kolmogorov-Uspensky machines (KU machines), Schönhage Storage Modification Machines (SMM), and linking automata as defined by Knuth. The work of Gandy and Markov are also described as influential precursors. Gurevich offers a 'strong' definition of an algorithm (boldface added):

> "...Turing's informal argument in favour of his thesis justifies a stronger thesis: every algorithm can be simulated by a Turing machine....In practice, it would be ridiculous...[Nevertheless,] [c]an one generalize Turing machines so that any algorithm, never mind how abstract, can be modeled by a generalized machine?...But suppose such generalized Turing machines exist. What would their states be?...a first-order structure ... a particular small instruction set suffices in all cases ... computation as an evolution of the state ... could be nondeterministic... can interact with their environment ... [could be] parallel and multi-agent ... [could have] dynamic semantics ... [the two underpinings of their work are:] Turing's thesis ...[and] the notion of (first order) structure of [Tarski 1933]" (Gurevich 2000, p. 1-2)

The above phrase computation as an evolution of the state differs markedly from the definition of Knuth and Stone—the "algorithm" as a Turing machine programme. Rather, it corresponds to what Turing called *the complete configuration* (cf Turing's definition in Undecidable, p. 118) — and includes *both* the current instruction (state) *and* the status of the tape. [cf Kleene (1952) p. 375 where he shows an example of a tape with 6 symbols on it—all other squares are blank—and how to Gödelize its combined table-tape status]. In Algorithm examples we see the evolution of the state first-hand.

1995 – DANIEL DENNETT: EVOLUTION AS AN ALGORITHMIC PROCESS

Philosopher Daniel Dennett analyses the importance of evolution as an algorithmic process in his 1995 book *Darwin's Dangerous Idea*. Dennett identifies three key features of an algorithm:

- Substrate neutrality: an algorithm relies on its *logical* structure. Thus, the particular form in which an algorithm is manifested is not important (Dennett's example is long division: it works equally well on paper, on parchment, on a computer screen, or using neon lights or in skywriting). (p. 51)
- Underlying mindlessness: no matter how complicated the end-product of the algorithmic process may be, each step in the algorithm is sufficiently simple to be performed by a non-sentient, mechanical device. The algorithm does not require a "brain" to maintain or operate it. "The standard textbook analogy notes that algorithms are *recipes* of sorts, designed to be followed by *novice* cooks."(p. 51)
- Guaranteed results: If the algorithm is executed correctly, it will always produce the same results. "An algorithm is a foolproof recipe."(p. 51)

It is on the basis of this analysis that Dennett concludes that "According to Darwin, evolution is an algorithmic process" (p. 60). However, in the previous page he has gone out on a much-further limb. In the context of his chapter titled "Processes as Algorithms" he states:

"But then . . are there any limits at all on what may be considered an algorithmic process? I guess

> the answer is NO; if you wanted to, you can treat any process at the abstract level as an algorithmic process. . . If what strikes you as puzzling is the uniformity of the [ocean's] sand grains or the strength of the [tempered-steel] blade, an algorithmic explanation is what will satisfy your curiosity — and it will be the truth. . . .
>
> "No matter how impressive the products of an algorithm, the underlying process always consists of nothing but a set of individualy [*sic*] mindless steps succeeding each other without the help of any intelligent supervision; they are 'automatic' by definition: the workings of an automaton." (p. 59)

It is unclear from the above whether Dennett is stating that the physical world by itself and without observers is intrinsically algorithmic (computational) or whether a symbol-processing observer is what is adding "meaning" to the observations.

2002 JOHN SEARLE ADDS A CLARIFYING CAVEAT TO DENNETT'S CHARACTERIZATION

John R. Searle and Daniel Dennett having been poking at one-another's philosophies of mind (cf philosophy of mind) for the past 30 years. Dennett hews to the Strong AI point of view that the logical structure of an algorithm is sufficient to explain mind; Searle, of Chinese room fame claims that logical structure is not sufficient, rather that: "Syntax [i.e. logical structure] is by itself not sufficient for semantic content [i.e. meaning]" (italics in original, Searle 2002:16). In other words, the "meaning" of symbols is relative to the mind that is using them; an algorithm—a logical construct—by itself is insufficient for a mind. Searle urges a note of caution to those who want to define algorithmic (computational) processes as intrinsic to nature (e.g. cosmology, physics, chemistry, etc.):

> "Computation . . . is observer-relative, and this is because computation is defined in terms of symbol manipulation, but the notion of a 'symbol' is not a notion of physics or chemistry. Something is a symbol only if it is used, treated or regarded as a symbol. The chinese room argument showed that semantics is not intrinsic to syntax. But what this shows is that syntax is not intrinsic to physics. . . . Something is a symbol only relative to some observer, user or agent who assigns a symbolic interpretation to it. . . you can assign a computational interpretation to anything. But if the question asks, 'Is consciousness intrinsically computational?' the answer is: *nothing is intrinsically computational.* Computation exists only relative to some agent or observer who imposes a computational interpretation on some phenomenon. This is an obvious point. I should have seen it ten years ago but I did not." (italics added, p. 17)

2002: BOOLOS-BURGESS-JEFFREY SPECIFICATION OF TURING MACHINE CALCULATION

An example in Boolos-Burgess-Jeffrey (2002) (pp. 31–32) demonstrates the precision required in a complete specification of an algorithm, in this case to add two numbers: m+n. It is similar to the Stone requirements above.

(i) They have discussed the role of "number format" in the computation and selected the "tally notation" to represent numbers:

> "Certainly computation can be harder in practice with some notations than others... But... it is possible in principle to do in any other notation, simply by translating the data... For purposes of framing a rigorously defined notion of computability, it is convenient to use monadic or tally notation" (p. 25-26)

(ii) At the outset of their example they specify the machine to be used in the computation as a Turing machine. They have previously specified (p. 26) that the Turing-machine will be of the 4-tuple, rather than 5-tuple, variety. For more on this convention see Turing machine.

(iii) Previously the authors have specified that the tape-head's position will be indicated by a subscript to the *right* of the scanned symbol. For more on this convention see Turing machine. (In the following, boldface is added for emphasis):

> "We have not given an official definition of what it is for a numerical function to be computable by a Turing machine, specifying how inputs or arguments are to be represented on the machine, and how outputs or values represented. Our specifications for a k-place function from positive integers to positive integers are as follows:

"(a) [Initial number format:] The arguments m_1, ... m_k, ... will be represented in monadic [unary] notation by blocks of those numbers of strokes, each block separated from the next by a single blank, on an otherwise blank tape.

Example: 3+2, 111B11

"(b) [Initial head location, initial state:] Initially, the machine will be scanning the leftmost 1 on the tape, and will be in its initial state, state 1.

Example: 3+2, $1_1$111B11

"(c) [Successful computation — number format at Halt:] If the function to be computed assigns a value n to the arguments that are represented initially on the tape, then the machine will eventually halt on a tape containing a block of strokes, and otherwise blank...

Example: 3+2, 11111

"(d) [Successful computation — head location at Halt:] In this case [c] the machine will halt scanning the left-most 1 on the tape...

Example: 3+2, 1_n1111

"(e) [Unsuccessful computation — failure to Halt or Halt with non-standard number format:] If the function that is to be computed assigns no value to the arguments that are represented initially on the tape, then the machine either will never halt, or will halt in some nonstandard configuration..."(ibid)

Example: B_n11111 or B1$1_n$111 or B1111$1_n$

This specification is incomplete: it requires the location of where the instructions are to be placed and their format in the machine—

(iv) in the finite state machine's TABLE or, in the case of a Universal Turing machine on the tape, and

(v) the Table of instructions in a specified format.

This later point is important. Boolos-Burgess-Jeffrey give a demonstration (p. 36) that the predictability of the entries in the table allow one to "shrink" the table by putting the entries in sequence and omitting the input state and the symbol.

2006: SIPSER'S ASSERTION AND HIS THREE LEVELS OF DESCRIPTION

For examples of this specification-method applied to the addition algorithm "m+n" see Algorithm examples. Sipser begins by defining '"algorithm" as follows:

> "Informally speaking, an *algorithm* is a collection of simple instructions for carrying out some task. Commonplace in everyday life, algorithms sometimes are called *procedures* or *recipes* (italics in original, p. 154)
>
> "...our real focus from now on is on algorithms. That is, the Turing machine merely serves as a precise model for the definition of algorithm .... we need only to be comfortable enough with Turing machines to believe that they capture all algorithms" (p. 156)

Does Sipser mean that "algorithm" is just "instructions" for a Turing machine, or is the combination of "instructions + a (specific variety of) Turing machine"? For example, he defines the two standard variants (multi-tape and nondeterministic) of his particular variant (not the same as Turing's original) and goes on, in his Problems (pages 160-161), to describes four more variants (write-once, doubly-infinite tape (i.e. left- and right-infinite), left reset, and "stay put instead of left). In addition, he imposes some constraints. First, the input must be encoded as a string (p. 157) and says of numeric encodings in the context of complexity theory:

> "But note that unary notation for encoding numbers (as in the number 17 encoded by the uary string 11111111111111111) isn't reasonable because it is exponentially larger than truly reasonable encodings, such as base *k* notation for any *k* e" 2."(p. 259)

van Emde Boas comments on a similar problem with respect to the random access machine (RAM) abstract model of computation sometimes used in place of the Turing machine when doing "analysis of algorithms": "The absence or presence of multiplicative and parallel bit manipulation operations is of relevance for the correct understanding of some results in the analysis of algorithms.

> ". . . [T]here hardly exists such as a thing as an "innocent" extension of the standard RAM model in the uniform time measures; either one only has additive arithmetic or one might as well include all reasonable multiplicative and/or bitwise Boolean instructions on small operands." (van Emde Boas, 1990:26)

With regards to a "description language" for algorithms Sipser finishes the job that Stone and Boolos-Burgess-

Jeffrey started (boldface added). He offers us three levels of description of Turing machine algorithms (p. 157): High-level description: "wherein we use ... prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head." Implementation description: "in which we use ... prose to describe the way that the Turing machine moves its head and the way that it stores data on its tape. At this level we do not give details of states or transition function." Formal description: "... the lowest, most detailed, level of description... that spells out in full the Turing machine's states, transition function, and so on."

2

Computer Algorithms

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended “target” computer(s), in order for the software on the target machines to *do something*. For instance, if a person is writing software that is supposed to print out a PDF document located at the operating system folder “/My Documents” at computer drive “D:” every Friday, they will write an algorithm that specifies the following actions: “If today’s date (computer time) is ‘Friday,’ open the document at ‘D:/My Documents’ and call the ‘print’ function”.

While this simple algorithm does not look into whether the printer has enough paper or whether the document has been moved into a different location, one can make this algorithm more robust and anticipate these problems by rewriting it as a formal CASE statement or as a (carefully crafted) sequence of IF-THEN-ELSE statements. For example the CASE statement might appear as follows (there are other possibilities):

- *Case* 1: IF today's date is NOT Friday THEN *exit this CASE instruction* ELSE
- *Case* 2: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is paper in the printer THEN print the document (and *exit this CASE instruction*) ELSE
- *Case* 3: IF today's date is Friday AND the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message (and *exit this CASE instruction*) ELSE
- *Case* 4: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is NO paper in the printer THEN, (i) display 'out of paper' error message (ii) *exit*. Note that CASE 3 includes two possibilities: (i) the document is NOT located at 'D:/My Documents' AND there's paper in the printer OR (ii) the document is NOT located at 'D:/My Documents' AND there's NO paper in the printer.

The sequence of IF-THEN-ELSE tests might look like this:

- *Test* 1: IF today's date is NOT Friday THEN *done* ELSE TEST 2:
- *Test* 2: IF the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message ELSE TEST 3:
- *Test* 3: IF there is NO paper in the printer THEN display 'out of paper' error message ELSE print the document.

These examples' logic grants precedence to the instance of "NO document at 'D:/My Documents' ".

ANALYSIS OF ALGORITHMS

PROGRAMS

When analyzing a Programme in terms of efficiency, we want to look at questions such as, "How long does it take for the Programme to run?" and "Is there another approach that will get the answer more quickly?" Efficiency will always be less important than correctness; if you don't care whether a Programme works correctly, you can make it run very quickly indeed, but no one will think it's much of an achievement! On the other hand, a Programme that gives a correct answer after ten thousand years isn't very useful either, so efficiency is often an important issue.

The term "efficiency" can refer to efficient use of almost any resource, including time, computer memory, disk space, or network bandwidth. In this section, however, we will deal exclusively with time efficiency, and the major question that we want to ask about a Programme is, how long does it take to perform its task?

It really makes little sense to classify an individual Programme as being "efficient" or "inefficient." It makes more sense to compare two (correct) Programmes that perform the same task and ask which one of the two is "more efficient," that is, which one performs the task more quickly. However, even here there are difficulties.

The running time of a Programme is not well-defined. The run time can be different depending on the number and speed of the processors in the computer on which it is run and, in the case of Java, on the design of the Java Virtual Machine which is used to interpret the Programme.

It can depend on details of the compiler which is used to translate the Programme from high-level language to machine language. Furthermore, the run time of a Programme depends on the size of the problem which the Programme has to solve. It takes a sorting Programme longer to sort 10000 items than it takes it to sort 100 items.

When the run times of two Programmes are compared, it often happens that Programme A solves small problems faster than Programme B, while Programme B solves large problems faster than Programme A, so that it is simply not the case that one Programme is faster than the other in all cases.

In spite of these difficulties, there is a field of computer science dedicated to analyzing the efficiency of Programmes. The field is known as Analysis of Algorithms. The focus is on algorithms, rather than on Programmes as such, to avoid having to deal with multiple implementations of the same algorithm written in different languages, compiled with different compilers, and running on different computers. Analysis of Algorithms is a mathematical field that abstracts away from these down-and-dirty details.

Still, even though it is a theoretical field, every working Programmemer should be aware of some of its techniques and results. This section is a very brief introduction to some of those techniques and results. Because this is not a mathematics book, the treatment will be rather informal. One of the main techniques of analysis of algorithms is asymptotic analysis. The term “asymptotic” here means basically “the tendency in the long run.” An asymptotic analysis of an algorithm’s run time looks at the question of how the run time depends on the size of the problem.

The analysis is asymptotic because it only considers what happens to the run time as the size of the problem increases without limit; it is not concerned with what happens for problems of small size or, in fact, for problems of any fixed finite size. Only what happens in the long run, as the problem size increases without limit, is important.

Showing that Algorithm A is asymptotically faster than Algorithm B doesn't necessarily mean that Algorithm A will run faster than Algorithm B for problems of size 10 or size 1000 or even size 1000000 — it only means that if you keep increasing the problem size, you will eventually come to a point where Algorithm A is faster than Algorithm B. An asymptotic analysis is only a first approximation, but in practice it often gives important and useful information.

Using this notation, we might say, for example, that an algorithm has a running time that is $O(n^2)$ or O(n) or O(log(n)). These notations are read "Big-Oh of n squared," "Big-Oh of n," and "Big-Oh of log n" (where log is a logarithm function). More generally, we can refer to O(f(n)) ("Big-Oh of f of n"), where f(n) is some function that assigns a positive real number to every positive integer n. The "n" in this notation refers to the size of the problem.

Before you can even begin an asymptotic analysis, you need some way to measure problem size. Usually, this is not a big issue. For example, if the problem is to sort a list of items, then the problem size can be taken to be the number of items in the list. When the input to an algorithm is an integer, as in the case of an algorithm that checks whether a given positive integer is prime, the usual measure of the size of a problem is the number of bits in the input integer rather than the integer itself. More generally, the number of bits in the input to a problem is often a good measure of the size of the problem.

To say that the running time of an algorithm is O(f(n)) means that for large values of the problem size, n, the running time of the algorithm is no bigger than some constant times f(n). (More rigorously, there is a number C and a positive integer M such that whenever n is greater than M, the run time is less than or equal to C*f(n).) The constant takes into account details such as the speed of the computer on which the algorithm is run; if you use a slower computer, you might have to use a bigger constant in the formula, but changing the constant won't change the basic fact that the run time is O(f(n)).

The constant also makes it unnecessary to say whether we are measuring time in seconds, years, CPU cycles, or any other unit of measure; a change from one unit of measure to another is just multiplication by a constant. Note also that O(f(n)) doesn't depend at all on what happens for small problem sizes, only on what happens in the long run as the problem size increases without limit.

To look at a simple example, consider the problem of adding up all the numbers in an array. The problem size, n, is the length of the array. Using A as the name of the array, the algorithm can be expressed in Java as:

```
total = 0;
for (int i = 0; i < n; i++)
total = total + A[i];
```

This algorithm performs the same operation, total = total + A[i], n times. The total time spent on this operation is a*n, where a is the time it takes to perform the operation once.

Now, this is not the only thing that is done in the algorithm. The value of i is incremented and is compared to n each time through the loop.

This adds an additional time of b*n to the run time, for some constant b. Furthermore, i and total both have to be initialized to zero; this adds some constant amount c to the running time.

The exact running time would then be (a+b)*n+c, where the constants a, b, and c depend on factors such as how the code is compiled and what computer it is run on. Using the fact that c is less than or equal to c*n for any positive integer n, we can say that the run time is less than or equal to (a+b+c)*n. That is, the run time is less than or equal to a constant times n. By definition, this means that the run time for this algorithm is O(n).

If this explanation is too mathematical for you, we can just note that for large values of n, the c in the formula (a+b)*n+c is insignificant compared to the other term, (a+b)*n. We say that c is a "lower order term." When doing asymptotic analysis, lower order terms can be discarded. A rough, but correct, asymptotic analysis of the algorithm would go something like this: Each iteration of the for loop takes a certain constant amount of time. There are n iterations of the loop, so the total run time is a constant times n, plus lower order terms (to account for the initialization). Disregarding lower order terms, we see that the run time is O(n).

Note that to say that an algorithm has run time O(f(n)) is to say that its run time is no bigger than some constant times f(n) (for large values of n). O(f(n)) puts an upper limit on the run time. However, the run time could be smaller, even much smaller. For example, if the run time is O(n), it would also be correct to say that the run time is O(n^2) or even O(n^{10}). If the run time is less than a constant times n, then it is certainly less than the same constant times n^2 or n^{10}.

Of course, sometimes it's useful to have a lower limit on the run time. That is, we want to be able to say that the run time is greater than or equal to some constant times f(n) (for large values of n). The notation for this is Ω(f(n)), read "Omega of f of n." "Omega" is the name of a letter in the Greek alphabet, and Ω is the upper case version of that letter. (To be technical, saying that the run time of an algorithm is Ω(f(n)) means that there is a positive number C and a positive integer M such that whenever n is greater than M, the run time is greater than or equal to C*f(n).) O(f(n)) tells you something about the maximum amount of time that you might have to wait for an algorithm to finish; Ω(f(n)) tells you something about the minimum time.

The algorithm for adding up the numbers in an array has a run time that is Ω(n) as well as O(n). When an algorithm has a run time that is both Ω(f(n)) and O(f(n)), its run time is said to be Θ(f(n)), read "Theta of f of n." (Theta is another letter from the Greek alphabet.) To say that the run time of an algorithm is Θ(f(n)) means that for large values of n, the run time is between a*f(n) and b*f(n), where a and b are constants (with b greater than a, and both greater than 0).

Let's look at another example. Consider the algorithm that can be expressed in Java in the following method:

```
 /**
 * Sorts the n array elements A[0], A[1]...,        A[n-1]
into increasing order.
 */
 public static simpleBubbleSort(int[] A, int        n) {
  for (int i = 0; i < n; i++) {
 // Do n passes through the array...
  for (int j = 0; j < n-1; j++) {
   if (A[j] > A[j+1]) {
 // A[j] and A[j+1] are out of order, so swap       them
     int temp = A[j];
     A[j] = A[j+1];
     A[j+1] = temp;
   }
  }
 }
}
```

Here, the parameter n represents the problem size. The outer for loop in the method is executed n times. Each time the outer for loop is executed, the inner for loop is exectued n-1 times, so the if statement is executed n*(n-1) times. This is n^2-n, but since lower order terms are not significant in an asymptotic analysis, it's good enough to say that the if statement is executed about n^2 times.

In particular, the test A[j] > A[j+1] is executed about n^2times, and this fact by itself is enough to say that the run time of the algorithm is W(n^2), that is, the run time is at least some constant times n^2. Furthermore, if we look at other operations — the assignment statements, incrementing i and j, etc. — none of them are executed more than n^2 times, so the run time is also O(n^2), that is, the run time is no more than some constant times n^2. Since it is both W(n^2) and O(n^2), the run time of the simpleBubbleSort algorithm is Q(n^2).

You should be aware that some people use the notation O(f(n)) as if it meant Θ(f(n)). That is, when they say that the run time of an algorithm is O(f(n)), they mean to say that the run time is about equal to a constant times f(n). For that, they should use Θ(f(n)). Properly speaking, O(f(n)) means that the run time is less than a constant times f(n), possibly much less.

So far, the analysis has ignored an important detail. We have looked at how run time depends on the problem size, but in fact the run time usually depends not just on the size of the problem but on the specific data that has to be processed. For example, the run time of a sorting algorithm can depend on the initial order of the items that are to be sorted, and not just on the number of items.

To account for this dependency, we can consider either the worst case run time analysis or the average case run time analysis of an algorithm. For a worst case run time analysis, we consider all possible problems of size n and look at the longest possible run time for all such problems. For an average case analysis, we consider all possible problems of size n and look at the average of the run times for all such problems. Usually, the average case analysis assumes that all problems of size n are equally likely to be encountered, although this is not always realistic — or even possible in the case where there is an infinite number of different problems of a given size.

In many cases, the average and the worst case run times are the same to within a constant multiple. This means that as far as asymptotic analysis is concerned, they are the same. That is, if the average case run time is O(f(n)) or Θ(f(n)), then so is the worst case. However, later in the book, we will encounter a few cases where the average and worst case asymptotic analyses differ. So, what do you really have to know about analysis of algorithms to read the rest of this book? We will not do any rigorous mathematical analysis, but you should be able to follow informal discussion of simple cases such as the examples that we have looked at in this section.

Most important, though, you should have a feeling for exactly what it means to say that the running time of an algorithm is $O(f(n))$ or $\Theta(f(n))$ for some common functions $f(n)$. The main point is that these notations do not tell you anything about the actual numerical value of the running time of the algorithm for any particular case.

They do not tell you anything at all about the running time for small values of n. What they do tell you is something about the rate of growth of the running time as the size of the problem increases.

Suppose you compare two algorithms that solve the same problem. The run time of one algorithm is $\Theta(n^2)$, while the run time of the second algorithm is $\Theta(n^3)$. What does this tell you? If you want to know which algorithm will be faster for some particular problem of size, say, 100, nothing is certain. As far as you can tell just from the asymptotic analysis, either algorithm could be faster for that particular case — or in any particular case. But what you can say for sure is that if you look at larger and larger problems, you will come to a point where the $\Theta(n^2)$ algorithm is faster than the $\Theta(n^3)$ algorithm. Furthermore, as you continue to increase the problem size, the relative advantage of the $\Theta(n^2)$ algorithm will continue to grow. There will be values of n for which the $\Theta(n^2)$ algorithm is a thousand times faster, a million times faster, a billion times faster, and so on. This is because for any positive constants a and b, the function $a*n^3$ grows faster than the function $b*n^2$ as n gets larger. (Mathematically, the limit of the ratio of $a*n^3$ to $b*n^2$ is infinite as n approaches infinity.)

This means that for "large" problems, a $\Theta(n^2)$ algorithm will definitely be faster than a $\Theta(n^3)$ algorithm. You just don't know — based on the asymptotic analysis alone — exactly how large "large" has to be. In practice, in fact, it is likely that the $\Theta(n^2)$ algorithm will be faster even for fairly small values of n, and absent other information you would generally prefer a $\Theta(n^2)$ algorithm to a $\Theta(n^3)$ algorithm.

So, to understand and apply asymptotic analysis, it is essential to have some idea of the rates of growth of some common functions. For the power functions n, n^2, n^3, n^4..., the larger the exponent, the greater the rate of growth of the function.

Exponential functions such as 2^n and 10^n, where the n is in the exponent, have a growth rate that is faster than that of any power function. In fact, exponential functions grow so quickly that an algorithm whose run time grows exponentially is almost certainly impractical even for relatively modest values of n, because the running time is just too long.

Another function that often turns up in asymptotic analysis is the logarithm function, log(n). There are actually many different logarithm functions, but the one that is usually used in computer science is the so-called logarithm to the base two, which is defined by the fact that $\log(2^x) = x$ for any number x. (Usually, this function is written $\log_2(n)$, but I will leave out the subscript 2, since I will only use the base-two logarithm in this book.)

The logarithm function grows very slowly. The growth rate of log(n) is much smaller than the growth rate of n. The growth rate of n*log(n) is a little larger than the growth rate of n, but much smaller than the growth rate of n^2.

POINTS

- *Introduction*: Analysis of Selection Sort
- *Introduction*: Analysis of Merge Sort
- Asymptotic Notation
- Asymptotic Notation Continued
- Heapsort
- Heapsort Continued
- Priority Queues (more heaps)
- Quicksort
- Bounds on Sorting and Linear Time Sorts
- Stable Sorts and Radix Sort
- Begin Dynamic Programmeming
- More Dynamic Programmeming
- *Begin Greedy Algorithms*: Huffman's Algorithm
- Dÿkstra's Algorithm
- Beyond Asymptotic Analysis: Memory Access Time
- B-Trees
- More B-Trees: Insertion and Splitting
- Union/Find
- Warshall's Algorithm, Floyd's Algorithm
- Large Integer Arithmetic
- RSA Public-Key Cryptosystem
- Begin Algorithms and Structural Complexity Theory
- Continue Algorithms and Structural Complexity Theory

- End Algorithms and Structural Complexity Theory
- Generating Permutations and Combinations
- Exam review with sample questions and solutions

MASTERS THEOREM

INTRODUCTION

In the analysis of algorithms, the master theorem, which is a specific case of the Akra-Bazzi theorem, provides a cookbook solution in asymptotic terms for recurrence relations of types that occur in practice.

It was popularized by the canonical algorithms which introduces and proves Nevertheless, not all recurrence relations can be solved with the use of the master theorem.

Consider a problem that can be solved using recurrence algorithm such as below:

```
  procedure T( n: size of problem)
defined  as:
   ifn<kthen exit
  Do work of amount f(n)
  T(n/b)
  T(n/b)
  repeat for a total of a times...
  T(n/b)
  end procedure
```

In above algorithm we are dividing the problem in to number of sub problems recursively, each sub problem being of size n/b. This can be visualized as building a call tree with each node of a tree an instance of one recursive call and its child nodes being instance of next calls. In above example, each node would have a number of child nodes.

Each node does amount of work that depends on size of sub problem n passed to that instance of recursive call and given by *f*(*n*). For example, if each recursive call is doing sorting then size of work does by each node in the tree would be at least *O*(*nlog*(*n*)). Total size of work done by entire tree is sum of work performed by all the nodes in the tree.

Algorithm such as above can be represented as recurrence relationship,

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)\,.$$

This recursive relationship can be successively substituted in to itself and expanded to obtain expression for total amount of work done[

Original Master theorem allows to easily calculate run time of such a recursive algorithm in Big O notation without doing expansion of above recursive relationship. A generalized form of Master Theorem by Akra and Bazzi introduced in 1998 is more usable, simpler and applicable on wide number of cases that occurs in practice.

The master theorem concerns recurrence relations of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{Where} a \geq 1, b > 1.$$

In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- $f(n)$ is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

GENERIC FORM

If it is true that $f(n) = O\left(n^{\log_b(a)-\epsilon}\right)$ for some constant ε > 0 (using Big O notation)

it follows that:

$$T(n) = \Theta\left(n^{\log_b a}\right)$$

EXAMPLE

$$T(n) = 8T\left(\frac{n}{2}\right) + 1000n^2$$

As one can see in the formula above, the variables get the following values:

$$a = 8, b = 2, f(n) = 1000n^2, \log_b a = \log_2 8 = 3$$

Now we have to check that the following equation holds:

$$f(n) = O\left(n^{\log a-\epsilon}\right)$$

$$1000n^2 = O\left(n^{3-\epsilon}\right)$$

If we choose ε = 1, we get:

$$1000n^2 = O\left(n^{3-1}\right) = 0\left(n^2\right)$$

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

$$T(n) = \Theta\left(n^{\log_b - a}\right)$$

If we insert the values from above, we finally get:

$$T(n) = \Theta\left(n^3\right)$$

Thus the given recurrence relation *T*(n) was in Θ(n^3).

CASE 2

GENERIC FORM

If it is true, for some constant k e" 0, that:

$$f(n) = \Theta\left(n^{\log_b a} \log^k n\right)$$

it follows that:

$$T(n) = \Theta\left(n^{\log_b a} \log^{k+1} n\right)$$

EXAMPLE

$$T(n) = 2T\left(\frac{n}{2}\right) + 10n$$

As we can see in the formula above the variables get the following values:

$$a = 2,\ b = 2,\ k = 0,\ f(n) = 10n,\ \log_b a = \log_2 2 = 1$$

Now we have to check that the following equation holds (in this case k=0):

$$f(n) = \Theta\left(n^{\log_b a}\right)$$

If we insert the values from above, we get:

$$10n = \Theta\left(n^1\right) = \Theta(n)$$

Since this equation holds, the second case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

$$T(n) = \Theta(n \log n)$$

If we insert the values from above, we finally get:

$$T(n) = \Theta(n \log n)$$

Thus the given recurrence relation $T(n)$ was in $\Theta(n \log n)$.

DESIGNING OF ALGORITHMS

OVERVIEW

At first glance, the algorithms move-until-out and quick-sort have little in common. One processes structures; the other processes lists. One creates a new structure for the generative step; the other splits up a list into three pieces and recurs on two of them. In short, a comparison of the two examples of generative recursion suggests that the design of algorithms is an ad hoc activity and that it is impossible to come up with a general design recipe. A closer look, however, suggests a different picture.

First, even though we speak of algorithms as processes that solve problems, they are still functions that consume and produce data. In other words, we still choose data to represent a problem, and we must definitely understand the nature of our data if we wish to understand the process. Second, we describe the processes in terms of data, for example, "creating a new structure" or "partitioning a list of numbers."

Third, we always distinguish between input data for which it is trivial to produce a solution and those for which it is not. Fourth, the generation of problems is the key to the design of algorithms. Although the idea of how to generate a new problem might be independent of a data representation, it must certainly be implemented for whatever form of representation we choose for our problem. Finally, once the generated problems have been solved, the solutions must be combined with other values.

Let us examine the six general stages of our structural design recipe in light of our discussion:

DATA ANALYSIS AND DESIGN

The choice of a data representation for a problem often affects our thinking about the process. Sometimes the description of a process dictates a particular choice of representation. On other occasions, it is possible and worthwhile to explore alternatives. In any case, we must Analyse and define our data collections.

CONTRACT, PURPOSE, HEADER

We also need a contract, a definition header, and a purpose statement. Since the generative step has no connection to the structure of the data definition, the purpose statement should not only specify what the function does but should also include a comment that explains in general terms how it works.

FUNCTION EXAMPLES

In our previous design recipes, the function examples merely specified which output the function should produce for some given input. For algorithms, examples should illustrate how the algorithm proceeds for some given input. This helps us to design, and readers to understand, the algorithm. For functions such as move-until-out the process is trivial and doesn't need more than a few words.

TEMPLATE

Our discussion suggests a general template for algorithms:

```
(define (generative-recursive-fun problem)
 cond
 [(trivially-solvable? problem)
 (determine-solution problem)]
 [else
 (combine-solutions
 ... problem...
    (generative-recursive-fun (generate-problem-
    1 problem))
      (generative-recursive-fun (generate-
problem-n problem)))]))
```

DEFINITION

This chapter is only a suggestive blueprint, not a definitive shape. Each function in the template is to remind us that we need to think about the following four questions:

- What is a trivially solvable problem?
- What is a corresponding solution?
- How do we generate new problems that are more easily solvable than the original problem? Is there

one new problem that we generate or are there several?

- Is the solution of the given problem the same as the solution of (one of) the new problems? Or, do we need to combine the solutions to create a solution for the original problem? And, if so, do we need anything from the original problem data?

To define the algorithm, we must express the answers to these four questions in terms of our chosen data representation.

TEST

Once we have a complete function, we must also test it. As before, the goal of testing is to discover bugs and to eliminate them. Remember that testing cannot validate that the function works correctly for all possible inputs.

TERMINATION

Unfortunately, the standard recipe is not good enough for the design of algorithms. Up to now, a function has always produced an output for any legitimate input. That is, the evaluation has always stopped. After all, by the nature of our recipe, each natural recursion consumes an immediate piece of the input, not the input itself. Because data is constructed in a hierarchical manner, this means that the input shrinks at every stage. Hence the function sooner or later consumes an atomic piece of data and stops.

With functions based on generative recursion, this is no longer true. The internal recursions don't consume an immediate component of the input but some new piece of data, which is generated from the input.

In addition, even the slightest mistake in translating the process description into a function definition may cause an infinite loop. The problem is most easily understood with an example. Consider the following definition of smaller-items, one of the two "problem generators" for quick-sort:

```
 ;; smaller-items: (listof number) number →
(listof number)
 ;; to create a list with all those numbers on
alon
 ;; that are smaller than or equal to threshold
 (define (smaller-items alon threshold)
  (cond
   [(empty? alon) empty]
   [else (if (⇐ (first alon) threshold)
         (cons (first alon) (smaller-items (rest
        alon) threshold))
         smaller-items (rest alon)
      threshold))]))
```

Instead of < it employs ⇐ to compare numbers. As a result, this function produces (list 5) when applied to (list 5) and 5.

The lesson from this example is that the design of algorithms requires one more step in our design recipe: a TERMINATION ARGUMENT, which explains why the process produces an output for every input and how the function implements this idea; or a warning, which explains when the process may not terminate.

For quick-sort, the argument might look like this:

At each step, quick-sort partitions the list into two sublists using smaller-items and larger-items. Each function produces a list that is smaller than the input (the second argument), even if the threshold (the first argument) is an item on the list. Hence each recursive application of quick-sortconsumes a strictly shorter list than the given one. Eventually, quick-sort receives and returns empty.

Without such an argument an algorithm must be considered incomplete. A good termination argument may on occasion also reveal additional termination cases. For example, (smaller-items N (list N)) and (larger-items N(list N)) always produce empty for any N. Therefore we know that quick-sort's answer for (list N) is (list N). To add this knowledge to quick-sort, we simply add a cond-clause:

```
(define (quick-sort alon)
(cond
  [(empty? alon) empty]
  [(empty? (rest alon)) alon]
  [else (append
          (quick-sort (smaller-items alon (first
          alon)))
            (list (first alon))
            (quick-sort (larger-items alon (first
            alon))))]))
```

The condition (empty? (rest alon)) is one way to ask whether alon contains one item.

EXAMPLE

Define the function tabulate-div, which accepts a number n and tabulates the list of all of its divisors, starting with 1 and ending in n. A number d is a divisior of a number n if the remainder of dividing n by d is 0, that is, (= (remainder n d) 0) is true. The smallest divisior of any number is 1; the largest one is the number itself.

EXERCISE

Develop the function merge-sort, which sorts a list of numbers in ascending order, using the following two auxiliary functions:

- The first one, make-singles, constructs a list of one-item lists from the given list of numbers. For example,
- (equal? (make-singles (list 2 5 9 3))
- (list (list 2) (list 5) (list 9) (list 3)))
- The second one, merge-all-Neighbours, merges pairs of Neighbouring lists. More specifically, it consumes a list of lists (of numbers) and merges Neighbours. For example,
- (equal? (merge-all-Neighbours (list (list 2) (list 5) (list 9) (list 3)))
- (list (list 2 5) (list 3 9)))
- (equal? (merge-all-Neighbours (list (list 2 5) (list 3 9)))
- (list (list 2 3 5 9)))

In general, this function yields a list that is approximately half as long as the input. Why is the output not always half as long as the input?

STRUCTURAL VERSUS GENERATIVE RECURSION

The template for algorithms is so general that it even covers functions based on structural recursion. Consider the version with one termination clause and one generation step:

```
(define (generative-recursive-fun problem)
   (cond
    [(trivially-solvable? problem)
    (determine-solution problem)]
   [else
      (combine-solutions
        problem
          (generative-recursive-fun (generate-
        problem problem)))]))
```

If we replace trivially-solvable? with empty? and generate-problem with rest, the outline is a template for a list-processing function:

```
(define (generative-recursive-fun problem)
   (cond
      [(empty? problem) (determine-solution
    problem)]
      [else
        (combine-solutions
         problem
          (generative-recursive-fun (rest
        problem)))]))
```

MAKING CHOICES

A user cannot distinguish sort and quick-sort. Both consume a list of numbers; both produce a list that consists of the same numbers arranged in ascending order. To an observer, the functions are completely equivalent. This raises the question of which of the two a Programmemer should provide. More generally, if we can develop a function using structural recursion and an equivalent one using generative recursion, what should we do? To understand this choice better, let's discuss another classical example of generative recursion from mathematics: the problem of finding the greatest common divisor of two positive natural numbers.

All such numbers have at least one divisor in common: 1. On occasion, this is also the only common divisor. For example, 2 and 3 have only 1 as common divisor because 2 and 3, respectively, are the only other divisors.

Then again, 6 and 25 are both numbers with several divisors:

- 6 is evenly divisible by 1, 2, 3, and 6;
- 25 is evenly divisible by 1, 5, and 25.

Still, the greatest common divisior of 25 and 6 is 1. In contrast, 18 and 24 have many common divisors:

- 18 is evenly divisible by 1, 2, 3, 6, 9, and 18;
- 24 is evenly divisible by 1, 2, 3, 4, 6, 8, 12, and 24.

The greatest common divisor is 6.

Following the design recipe, we start with a contract, a purpose statement, and a header:

```
;; gcd: N[⇒ 1] N[⇒ 1] → N
;; to find the greatest common divisior of n      and m
(define (gcd n m)
 ...)
```

The contract specifies the precise inputs: natural numbers that are greater or equal to 1 (not 0).

Now we need to make a decision whether we want to pursue a design based on structural or on generative recursion. Since the answer is by no means obvious, we develop both. For the structural version, we must consider which input the function should process: n, m, or both.

A moment's consideration suggests that what we really need is a function that starts with the smaller of the two and outputs the first number smaller or equal to this input that evenly divides both n and m.

```
 ;; gcd-structural: N[⇒ 1] N[⇒ 1] → N      ;; to find
the greatest common divisior of n       and m
 ;; structural recursion using data definition
of N[⇒ 1] (define (gcd-structural n m) (local
  ((define (first-divisior-⇐ i)
        (cond
          [(= i 1) 1]
          [else (cond
                  [(and (= (remainder n i) 0)
                        (= (remainder m i)
                         0)) i]
                   [else (first-divisior-⇐ (-
                  i 1))])]))))
                 (first-divisior-⇐ (min m
                  n))))
```

The conditions "evenly divisible" have been encoded as (= (remainder n i) 0) and (=(remainder m i) 0). The two ensure that i divides n and m without a remainder. Testing gcd-structural with the examples shows that it finds the expected answers. Although the design of gcd-structural is rather straightforward, it is also naive. It simply tests for every number whether it divides both n and m evenly and returns the first such number. For small natural numbers, this process works just fine.

Consider the following example, however:

```
(gcd-structural 101135853 45014640)
```

The result is 177 and to get there gcd-structural had to compare 101135676, that is, 101135853 – 177, numbers. This is a large effort and even reasonably fast computers spend several minutes on this task.

EXERCISES

Enter the definition of gcd-structural into the Definitions window and evaluate (time (gcd-structural 101135853 45014640)) in the Interactionswindow.

After testing gcd-structural conduct the performance tests in the Full Scheme language (without debugging), which evaluates expressions faster than the lower language levels but with less protection. Add (require-library "core.ss") to the top of the Definitions window. Have some reading handy!

Since mathematicians recognized the inefficiency of the "structural algorithm" a long time ago, they studied the problem of finding divisiors in more depth.

The essential insight is that for two natural numbers larger and smaller, their greatest common divisor is equal to the greatest common divisior of smaller and theremainder of larger divided into smaller.

Here is how we can put this insight into equational form:

```
  (gcd larger smaller)
 =(gcd smaller (remainder larger smaller))
```

Since (remainder larger smaller) is smaller than both larger and smaller, the right-hand side use of gcd consumes smaller first.

Here is how this insight applies to our small example:

- The given numbers are 18 and 24.
- According to the mathematicians' insight, they have the same greatest common divisor as 18 and 6.
- And these two have the same greatest common divisor as 6 and 0.

And here we seem stuck because 0 is nothing expected. But, 0 can be evenly divided by every number, so we have found our answer: 6.

Working through the example not only explains the idea but also suggests how to discover the case with a trivial solution. When the smaller of the two numbers is 0, the result is the larger number.

Putting everything together, we get the following definition:

```
 ;; gcd-generative: N[⇒ 1] N[⇒1] → N
 ;; to find the greatest common divisior of n      and m
 ;; generative recursion: (gcd n m) = (gcd n
(remainder m n)) if (⇐ m n)
  (define (gcd-generative n m)
   (local ((define (clever-gcd larger smaller)
           (cond
             [(= smaller 0) larger]
             [else (clever-gcd smaller
           (remainder larger smaller))])))
   (clever-gcd (max m n) (min m n))))
```

The local definition introduces the workhorse of the function: clever-gcd, a function based on generative recursion. Its first line discovers the trivially solvable case by comparing smaller to 0 and produces the matching solution.

The generative step uses smaller as the new first argument and (remainder largersmaller) as the new second argument to clever-gcd, exploiting the above equation.

If we now use gcd-generative with our complex example from above:

```
(gcd-generative 101135853 45014640)
```

we see that the response is nearly instantaneous. A hand-evaluation shows that clever-gcd recurs only nine times before it produces the solution: 177. In short, generative recursion has helped find us a much faster solution to our problem.

PROCESS

For the maximum subarray problem, if you didn't know any better, you'd probably implement a solution that Analyses every possible subarray, and returns the one with the maximum sum:

```
  class Array
   def sum
       result = 0
     self.each do |i|
            result+=i
     end
     return result
 end
#test every sub array - brute force!
 def max_sub_array_order_ncubed
       left_index = 0
       right_index = 0
       max_value = self [left_ index.. right_
     index].sum
       for i in (0..self.length)
            for j in (i..self.length
                this_value = self[i..j].sum
                if (this_value     >     max_value)
                max_value       =       this_value
                left_index=i
                right_index=j
             end
        end
   end
return  self[left_index..right_index]
      end
  end
```

If you were a bit more clever, you might notice that self[i..j].sum is equal toself[i..(j–1)].sum + self[j] in the innermost loop (the sum method itself), and use an accumulator there as opposed to calculating it each time. That takes you down from n^3 to n^2 time.

But there are (at least) two other ways to solve this problem:

- A divide and conquer approach that uses recursion and calculates the left and right maximum contiguous subarrays (MCS), along with the MCS that contains the right-most element in the left side and the left-most element in the right side. It compares the three and returns the one with the maximum sum. This gets us to O(n log n) time.
- An approach I'll call "expanding sliding window." If memory serves me correct, this aptly describes it or was the way a professor of mine described it. In any case, the "expanding sliding window" can do it in one pass (O(n) time), at the cost of a few more variables.

3

Computational Complexity Theory

Computational complexity theory is a branch of the theory of computation in theoretical computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty. In this context, a computational problem is understood to be a task that is in principle amenable to being solved by a computer (which basically means that the problem can be stated by a set of mathematical instructions). Informally, a computational problem consists of problem instances and solutions to these problem instances. For example, primality testing is the problem of determining whether a given number is prime or not. The instances of this problem are natural numbers, and the solution to an instance is *yes* or *no* based on whether the number is prime or not. A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication (used in communication complexity), the number of gates in a circuit (used in circuit complexity) and the number of processors (used in parallel computing). One of the roles of computational complexity theory is to determine the practical limits on what computers can and cannot do. Closely related fields in theoretical computer science are analysis of algorithms and computability theory. A key distinction between analysis of algorithms and computational complexity theory is that the former is devoted to analyzing the amount of resources needed by a particular algorithm to solve a problem, whereas the latter asks a more general question about all possible algorithms that could be used to solve the same problem. More precisely, it tries to classify problems that can or cannot be solved with appropriately restricted resources. In turn, imposing restrictions on the available resources is what distinguishes computational complexity from computability theory: the latter theory asks what kind of problems can be solved in principle algorithmically.

COMPUTATIONAL PROBLEMS

PROBLEM INSTANCES

A computational problem can be viewed as an infinite collection of *instances* together with a *solution* for every instance. The input string for a computational problem is referred to as a problem instance, and should not be confused with the problem itself. In computational complexity theory, a problem refers to the abstract question to be solved. In contrast, an instance of this problem is a rather concrete utterance, which can serve as the input for a decision problem. For example, consider the problem of primality testing. The instance is a number and the solution is "yes" if the number is prime and "no" otherwise. Alternately, the instance is a particular input to the problem, and the solution is the output corresponding to the given input. To further highlight the difference between a problem and an instance, consider the following instance of the decision version of the traveling salesman problem: Is there a route of length at most 2000 kilometres passing through all of Germany's 15 largest cities? The answer to this particular problem instance is of little use for solving other instances of the problem, such as asking for a round trip through all sites in Milan whose total length is at most 10 km. For this reason, complexity theory addresses computational problems and not particular problem instances.

REPRESENTING PROBLEM INSTANCES

When considering computational problems, a problem instance is a string over an alphabet. Usually, the alphabet is taken to be the binary alphabet (i.e., the set {0,1}), and thus the strings are bitstrings. As in a real-world computer, mathematical objects other than bitstrings must be suitably encoded. For example, integers can be represented in binary notation, and graphs can be encoded directly via their adjacency matrices, or by encoding their adjacency lists in binary. Even though some proofs of complexity-theoretic theorems regularly assume some concrete choice of input encoding, one tries to keep the discussion abstract enough to be independent of the choice of encoding. This can be achieved by ensuring that different representations can be transformed into each other efficiently.

DECISION PROBLEMS AS FORMAL LANGUAGES

Decision problems are one of the central objects of study in computational complexity theory. A decision problem is a special type of computational problem whose answer is either *yes* or *no*, or alternately either 1 or 0. A decision problem can be viewed as a formal language, where the members of the language are instances whose answer is yes, and the non-members are those instances whose output is no. The objective is to decide, with the aid of an algorithm, whether a given input string is member of the formal language under consideration. If the algorithm deciding this problem returns the answer *yes*, the algorithm is said to accept the input string, otherwise it is said to reject the input. An example of a decision problem is the following. The input is an arbitrary graph. The problem consists in deciding whether the given graph is connected, or not. The formal language associated with this decision problem is then the set of all connected graphs—of course, to obtain a precise definition of this language, one has to decide how graphs are encoded as binary strings.

FUNCTION PROBLEMS

A function problem is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just yes or no. Notable examples include the traveling salesman problem and the integer factorization problem. It is tempting to think that the notion of function problems is much richer than the notion of decision problems. However, this is not really the case, since function problems can be recast as decision problems. For example, the multiplication of two integers can be expressed as the set of triples (a, b, c) such that the relation $a \times b = c$ holds. Deciding whether a given triple is member of this set corresponds to solving the problem of multiplying two numbers.

MEASURING THE SIZE OF AN INSTANCE

To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem. However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve. Thus the time required to solve a problem (or the space required, or any measure of complexity) is calculated as function of the size of the instance. This is usually taken to be the size of the input in bits. Complexity theory is interested in how algorithms scale with an increase in the input size. For instance, in the problem of finding whether a graph is connected, how much more time does it take to solve a problem for a graph with $2n$ vertices compared to the time taken for a graph with n vertices? If the input size is n, the time taken can be expressed as a function of n. Since the time taken on different inputs of the same size can be different, the worst-case time complexity T(n) is defined to be the maximum time taken over all inputs of size n. If T(n) is a polynomial in n, then the algorithm is said to be a polynomial time algorithm. Cobham's thesis says that a problem can be solved with a feasible amount of resources if it admits a polynomial time algorithm.

MACHINE MODELS AND COMPLEXITY MEASURES

TURING MACHINE

A Turing machine is a mathematical model of a general computing machine. It is a theoretical device that manipulates symbols contained on a strip of tape. Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine. It is believed that if a problem can be solved by an algorithm, there exists a Turing machine that solves the problem. Indeed, this is the statement of the Church–Turing thesis. Furthermore, it is known that everything that can be computed on other models of computation known to us today, such as a RAM machine, Conway's Game of Life, cellular automata or any programming language can be computed on a Turing machine. Since Turing machines are easy to analyze mathematically, and are believed to be as powerful as any other model of computation, the Turing machine is the most commonly used model in complexity theory. Many types of Turing machines are used to define complexity classes, such as deterministic Turing machines, probabilistic Turing machines, non-deterministic Turing machines, quantum Turing machines, symmetric Turing machines and alternating Turing machines. They are all equally powerful in principle, but when resources (such as time or space) are bounded, some of these may be more powerful than others.

A deterministic Turing machine is the most basic Turing machine, which uses a fixed set of rules to determine its future actions. A probabilistic Turing machine is a deterministic Turing machine with an extra supply of random bits. The ability to make probabilistic decisions often helps algorithms solve problems more efficiently. Algorithms that use random bits are called randomized algorithms. A non-deterministic Turing machine is a deterministic Turing machine with an added feature of non-determinism, which allows a Turing machine to have multiple possible future actions from a given state. One way to view non-determinism is that the Turing machine branches into many possible computational paths at each step, and if it solves the problem in any of these branches, it is said to have solved the problem. Clearly, this model is not meant to be a physically realizable model, it is just a theoretically interesting abstract machine that gives rise to particularly interesting complexity classes. For examples, see nondeterministic algorithm.

OTHER MACHINE MODELS

Many machine models different from the standard multi-tape Turing machines have been proposed in the literature, for example random access machines. Perhaps surprisingly, each of these models can be converted to another without providing any extra computational power. The time and memory consumption of these alternate models may vary. What all these models have in common is that the machines operate deterministically. However, some computational problems are easier to analyze in terms of more unusual resources. For example, a nondeterministic Turing machine is a computational model that is allowed to branch out to check many different possibilities at once. The nondeterministic Turing machine has very little to do with how we physically want to compute algorithms, but its branching exactly captures many of the mathematical models we want to analyze, so that nondeterministic time is a very important resource in analyzing computational problems.

COMPLEXITY MEASURES

For a precise definition of what it means to solve a problem using a given amount of time and space, a computational model such as the deterministic Turing machine is used. The *time required* by a deterministic Turing machine M on input x is the total number of state transitions, or steps, the machine makes before it halts and outputs the answer ("yes" or "no"). A Turing machine M is said to operate within time $f(n)$, if the time required by M on each input of length n is at most $f(n)$. A decision problem A can be solved in time $f(n)$ if there exists a Turing machine operating in time $f(n)$ that solves the problem. Since complexity theory is interested in classifying problems based on their difficulty, one defines sets of problems based on some criteria. For instance, the set of problems solvable within time $f(n)$ on a deterministic Turing machine is then denoted by DTIME($f(n)$). Analogous definitions can be made for space requirements. Although time and space are the most well-known complexity resources, any complexity measure can be viewed as a computational resource. Complexity measures are very generally defined by the Blum complexity axioms. Other complexity measures used in complexity theory include communication complexity, circuit complexity, and decision tree complexity.

BEST, WORST AND AVERAGE CASE COMPLEXITY

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size n may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size n.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size n.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size n.

For example, consider the deterministic sorting algorithm quicksort. This solves the problem of sorting a list of integers that is given as the input. The best-case scenario is when the input is already sorted, and the algorithm takes time O(n log n) for such inputs. The worst-case is when the input is sorted in reverse order, and the algorithm takes time O(n^2) for this case. If we assume that all possible permutations of the input list are equally likely, the average time taken for sorting is O(n log n).

UPPER AND LOWER BOUNDS ON THE COMPLEXITY OF PROBLEMS

To classify the computation time (or similar resources, such as space consumption), one is interested in proving upper and lower bounds on the minimum amount of time required by the most efficient algorithm solving a given problem. The complexity of an algorithm is usually taken to be its worst-case complexity, unless specified otherwise. Analyzing a particular algorithm falls under the field of analysis of algorithms. To show an upper bound $T(n)$ on the time complexity of a problem, one needs to show only that there is a particular algorithm with running time at most $T(n)$. However, proving lower bounds is much more difficult, since lower bounds make a statement about all possible algorithms that solve a given problem. The phrase "all possible algorithms" includes not just the algorithms known today, but any algorithm that might be discovered in the future. To show a lower bound of $T(n)$ for a problem requires showing that no algorithm can have time complexity lower than $T(n)$. Upper and lower bounds are usually stated using the big Oh notation, which hides constant factors and smaller terms. This makes the bounds independent of the specific details of the computational model used. For instance, if $T(n) = 7n^2 + 15n + 40$, in big Oh notation one would write $T(n) = O(n^2)$.

COMPLEXITY CLASSES

DEFINING COMPLEXITY CLASSES

A complexity class is a set of problems of related complexity. Simpler complexity classes are defined by the following factors:

- The type of computational problem: The most commonly used problems are decision problems. However, complexity classes can be defined based on function problems, counting problems, optimization problems, promise problems, etc.
- The model of computation: The most common model of computation is the deterministic Turing machine, but many complexity classes are based on nondeterministic Turing machines, Boolean circuits, quantum Turing machines, monotone circuits, etc.
- The resource (or resources) that are being bounded and the bounds: These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth", etc.

Of course, some complexity classes have complex definitions that do not fit into this framework. Thus, a typical complexity class has a definition like the following:

The set of decision problems solvable by a deterministic Turing machine within time $f(n)$. (This complexity class is known as DTIME($f(n)$).) But bounding the computation time above by some concrete function $f(n)$ often yields complexity classes that depend on the chosen machine model. For instance, the language $\{xx \mid x$ is any binary string$\}$ can be solved in linear time on a multi-tape Turing machine, but necessarily requires quadratic time in the model of single-tape Turing machines. If we allow polynomial variations in running time, Cobham-Edmonds thesis states that "the time complexities in any two reasonable and general models of computation are polynomially related" (Goldreich 2008, Chapter 1.2).

This forms the basis for the complexity class P, which is the set of decision problems solvable by a deterministic Turing machine within polynomial time. The corresponding set of function problems is FP.

IMPORTANT COMPLEXITY CLASSES

Many important complexity classes can be defined by bounding the time or space used by the algorithm. Some important complexity classes of decision problems. Other important complexity classes include BPP, ZPP and RP, which are defined using probabilistic Turing machines; AC and NC, which are defined using Boolean circuits and BQP and QMA, which are defined using quantum Turing machines. #P is an important complexity class of counting problems (not decision problems). Classes like IP and AM are defined using Interactive proof systems. ALL is the class of all decision problems.

REDUCTION

Many complexity classes are defined using the concept of a reduction. A reduction is a transformation of one problem into another problem. It captures the informal notion of a problem being at least as difficult as another problem. For instance, if a problem *X* can be solved using an algorithm for *Y*, *X* is no more difficult than *Y*, and we say that *X reduces* to *Y*. There are many different types of reductions, based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions, and the bound on the complexity of reductions, such as polynomial-time reductions or log-space reductions. The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

This motivates the concept of a problem being hard for a complexity class. A problem *X* is *hard* for a class of problems *C* if every problem in *C* can be reduced to *X*. Thus no problem in *C* is harder than *X*, since an algorithm for *X* allows us to solve any problem in *C*. Of course, the notion of hard problems depends on the type of reduction being used. For complexity classes larger than P, polynomial-time reductions are commonly used. In particular, the set of problems that are hard for NP is the set of NP-hard problems. If a problem *X* is in *C* and hard for *C*, then *X* is said to be *complete* for *C*. This means that *X* is the hardest problem in *C*. (Since many problems could be equally hard, one might say that *X* is one of the hardest problems in *C*.) Thus the class of NP-complete problems contains the most difficult problems in NP, in the sense that they are the ones most likely not to be in P. Because the problem P = NP is not solved, being able to reduce a known NP-complete problem, $Đ_2$, to another problem, $Đ_1$, would indicate that there is no known polynomial-time solution for $Đ_1$.

This is because a polynomial-time solution to $Đ_1$ would yield a polynomial-time solution to $Đ_2$. Similarly, because all NP problems can be reduced to the set, finding an NP-complete problem that can be solved in polynomial time would mean that P = NP.

IMPORTANT OPEN PROBLEMS

P VERSUS NP PROBLEM

The complexity class P is often seen as a mathematical abstraction modeling those computational tasks that admit an efficient algorithm. This hypothesis is called the Cobham–Edmonds thesis. The complexity class NP, on the other hand, contains many problems that people would like to solve efficiently, but for which no efficient algorithm is known, such as the Boolean satisfiability problem, the Hamiltonian path problem and the vertex cover problem. Since deterministic Turing machines are special nondeterministic Turing machines, it is easily observed that each problem in P is also member of the class NP. The question of whether P equals NP is one of the most important open questions in theoretical computer science because of the wide implications of a solution. If the answer is yes, many important problems can be shown to have more efficient solutions. These include various types of integer programming problems in operations research, many problems in logistics, protein structure prediction in biology, and the ability to find formal proofs of pure mathematics theorems. The P versus NP problem is one of the Millennium Prize Problems proposed by the Clay Mathematics Institute. There is a US$1,000,000 prize for resolving the problem.

PROBLEMS IN NP NOT KNOWN TO BE IN P OR NP-COMPLETE

It was shown by Ladner that if P "" NP then there exist problems in NP that are neither in P nor NP-complete. Such problems are called NP-intermediate problems. The graph isomorphism problem, the discrete logarithm problem and the integer factorization problem are examples of problems believed to be NP-intermediate. They are some of the very few NP problems not known to be in P or to be NP-complete. The graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic. An important unsolved problem in complexity theory is whether the graph isomorphism problem is in P, NP-complete, or NP-intermediate. The answer is not known, but it is believed that the problem is at least not NP-complete. If graph isomorphism is NP-complete, the polynomial time hierarchy collapses to its second level.

Since it is widely believed that the polynomial hierarchy does not collapse to any finite level, it is believed that graph isomorphism is not NP-complete.

The best algorithm for this problem, due to Laszlo Babai and Eugene Luks has run time $2^{O(“(n \log n))}$ for graphs with n vertices. The integer factorization problem is the computational problem of determining the prime factorization of a given integer. Phrased as a decision problem, it is the problem of deciding whether the input has a factor less than k. No efficient integer factorization algorithm is known, and this fact forms the basis of several modern cryptographic systems, such as the RSA algorithm. The integer factorization problem is in NP and in co-NP (and even in UP and co-UP). If the problem is NP-complete, the polynomial time hierarchy will collapse to its first level (i.e., NP will equal co-NP). The best known algorithm for integer factorization is the general number field sieve, which takes time $O(e^{(64/9)1/3(n.\log 2)1/3(\log (n.\log 2))2/3})$ to factor an n-bit integer. However, the best known quantum algorithm for this problem, Shor’s algorithm, does run in polynomial time. Unfortunately, this fact doesn’t say much about where the problem lies with respect to non-quantum complexity classes.

SEPARATIONS BETWEEN OTHER COMPLEXITY CLASSES

Many known complexity classes are suspected to be unequal, but this has not been proved. For instance P” NP” PP” PSPACE, but it is possible that P = PSPACE. If P is not equal to NP, then P is not equal to PSPACE either. Since there are many known complexity classes between P and

PSPACE, such as RP, BPP, PP, BQP, MA, PH, etc., it is possible that all these complexity classes collapse to one class. Proving that any of these classes are unequal would be a major breakthrough in complexity theory. Along the same lines, co-NP is the class containing the complement problems (i.e. problems with the *yes*/*no* answers reversed) of NP problems. It is believed that NP is not equal to co-NP; however, it has not yet been proven. It has been shown that if these two complexity classes are not equal then P is not equal to NP. Similarly, it is not known if L (the set of all problems that can be solved in logarithmic space) is strictly contained in P or equal to P. Again, there are many complexity classes between the two, such as NL and NC, and it is not known if they are distinct or equal classes.

INTRACTABILITY

Problems that can be solved but not fast enough for the solution to be useful are called *intractable*. In complexity theory, problems that lack polynomial-time solutions are considered to be intractable for more than the smallest inputs. In fact, the Cobham–Edmonds thesis states that only those problems that can be solved in polynomial time can be feasibly computed on some computational device. Problems that are known to be intractable in this sense include those that are EXPTIME-hard. If NP is not the same as P, then the NP-complete problems are also intractable in this sense. To see why exponential-time algorithms might be unusable in practice, consider a programme that makes 2^n operations before halting. For small n, say 100, and assuming for the sake of example that the computer does

10^{12} operations each second, the programme would run for about 4×10^{10} years, which is roughly the age of the universe. Even with a much faster computer, the programme would only be useful for very small instances and in that sense the intractability of a problem is somewhat independent of technological progress. Nevertheless a polynomial time algorithm is not always practical.

If its running time is, say, n^{15}, it is unreasonable to consider it efficient and it is still useless except on small instances. What intractability means in practice is open to debate. Saying that a problem is not in P does not imply that all large cases of the problem are hard or even that most of them are. For example the decision problem in Presburger arithmetic has been shown not to be in P, yet algorithms have been written that solve the problem in reasonable times in most cases. Similarly, algorithms can solve the NP-complete knapsack problem over a wide range of sizes in less than quadratic time and SAT solvers routinely handle large instances of the NP-complete Boolean satisfiability problem.

CONTINUOUS COMPLEXITY THEORY

Continuous complexity theory can refer to complexity theory of problems that involve continuous functions that are approximated by discretizations, as studied in numerical analysis. One approach to complexity theory of numerical analysis is information based complexity. Continuous complexity theory can also refer to complexity theory of the use of analog computation, which uses continuous dynamical systems and differential equations. Control theory can be considered a form of computation and differential equations are used in the modelling of continuous-time and hybrid discrete-continuous-time systems.

HISTORY

Before the actual research explicitly devoted to the complexity of algorithmic problems started off, numerous foundations were laid out by various researchers. Most influential among these was the definition of Turing machines by Alan Turing in 1936, which turned out to be a very robust and flexible notion of computer. Fortnow & Homer (2003) date the beginning of systematic studies in computational complexity to the seminal paper "On the Computational Complexity of Algorithms" by Juris Hartmanis and Richard Stearns (1965), which laid out the definitions of time and space complexity and proved the hierarchy theorems. According to Fortnow & Homer (2003), earlier papers studying problems solvable by Turing machines with specific bounded resources include John Myhill's definition of linear bounded automata (Myhill 1960), Raymond Smullyan's study of rudimentary sets (1961), as well as Hisao Yamada's paper on real-time computations (1962). Somewhat earlier, Boris Trakhtenbrot (1956), a pioneer in the field from the USSR, studied another specific complexity measure. As he remembers:

> However, [my] initial interest [in automata theory] was increasingly set aside in favour of computational complexity, an exciting fusion of combinatorial methods, inherited from switching theory, with the conceptual arsenal of the theory

of algorithms. These ideas had occurred to me earlier in 1955 when I coined the term "signalizing function", which is nowadays commonly known as "complexity measure".

—Boris Trakhtenbrot, From Logic to Theoretical Computer Science – An Update. In: *Pillars of Computer Science*, LNCS 4800, Springer 2008.

In 1967, Manuel Blum developed an axiomatic complexity theory based on his axioms and proved an important result, the so called, speed-up theorem. The field really began to flourish when the US researcher Stephen Cook and, working independently, Leonid Levin in the USSR, proved that there exist practically relevant problems that are NP-complete.

In 1972, Richard Karp took this idea a leap forward with his landmark paper, "Reducibility Among Combinatorial Problems", in which he showed that 21 diverse combinatorial and graph theoretical problems, each infamous for its computational intractability, are NP-complete.

MODEL OF COMPUTATION

In computability theory and computational complexity theory, a model of computation is the definition of the set of allowable operations used in computation and their respective costs. It is used for measuring the complexity of an algorithm in execution time and or memory space: by assuming a certain model of computation, it is possible to analyze the computational resources required or to discuss the limitations of algorithms or computers.

Some examples of models include Turing machines, recursive functions, lambda calculus, and production systems. In model-driven engineering, the model of computation explains how the behaviour of the whole system is the result of the behaviour of each of its components. In the field of runtime analysis of algorithms, it is common to specify a computational model in terms of *primitive operations* allowed which have unit cost, or simply unit-cost operations.

A commonly used example is the random access machine, which has unit cost for read and write access to all of its memory cells. In this respect, it differs from the above mentioned Turing machine model.

There are many models of computation, differing in the set of admissible operations and their computations cost. They fall into the following broad categories: abstract machine, used in proofs of computability and upper bounds on computational complexity of algorithms, and decision tree models, used in proofs of lower bounds on computational complexity of algorithmic problems. A key point which is often overlooked is that published lower bounds for problems are often given for a model of computation that is more restricted than the set of operations that you could use in practice and therefore there are algorithms that are faster than what would naively be thought possible.

PARTIAL FUNCTION

In mathematics, a partial function from *X* to *Y* is a function *f: X' '! Y*, where *X'* is a subset of *X*. It generalizes the concept of a function by not forcing *f* to map *every* element of *X* to an element of *Y* (only some subset *X'* of *X*). If *X'* = *X*, then *f* is called a total function and is equivalent to a function. Partial functions are often used when the exact domain, *X*’, is not known (e.g. many functions in computability theory). Specifically, we will say that for any *x* ” *X*, either:

- *f*(*x*) = *y* ” *Y* (it is defined as a single element in *Y*) or
- *f*(*x*) is undefined.

DOMAIN OF A PARTIAL FUNCTION

There are two distinct meanings in current mathematical usage for the notion of the domain of a partial function. Most mathematicians, including recursion theorists, use the term “domain of *f*” for the set of all values *x* such that *f(x)* is defined (*X*’ above). But some, particularly category theorists, consider the domain of a partial function *f*:*X*’!*Y* to be *X*, and refer to *X*’ as the domain of definition. Occasionally, a partial function with domain *X* and codomain *Y* is written as *f*: *X* ø! *Y*, using an arrow with vertical stroke. A partial function is said to be injective or surjective when the total function given by the restriction of the partial function to its domain of definition is. A partial function may be both injective and surjective, but the term bijection generally only applies to total functions. An injective partial function may be inverted to an injective partial function, and a partial function which is both injective and surjective has an injective function as inverse.

DISCUSSION AND EXAMPLES

A partial function that is not a total function since the element 1 in the left-hand set is not associated with anything in the right-hand set.

NATURAL LOGARITHM

Consider the natural logarithm function mapping the real numbers to themselves. The logarithm of a non-positive real is not a real number, so the natural logarithm function doesn't associate any real number in the codomain with any non-positive real number in the domain. Therefore, the natural logarithm function is not a total function when viewed as a function from the reals to themselves, but it is a partial function. If the domain is restricted to only include the positive reals (that is, if the natural logarithm function is viewed as a function from the positive reals to the reals), then the natural logarithm is a total function.

BOTTOM TYPE

In some automated theorem proving systems a partial function is considered as returning the bottom type when it is undefined. The Curry-Howard correspondence uses this to map proofs and computer programmes to each other. In computer science a partial function corresponds to a subroutine that raises an exception or loops forever. The IEEE floating point standard defines a Not-a-number value which is returned when a floating point operation is undefined and exceptions are suppressed, e.g. when the square root of a negative number is requested.

4

Algorithmic Efficiency

In computer science, efficiency is used to describe properties of an algorithm relating to how much of various types of resources it consumes. Algorithmic efficiency can be thought of as analogous to engineering productivity for a repeating or continuous process, where the goal is to reduce resource consumption, including time to completion, to some acceptable, optimal level.

SOFTWARE METRICS

The two most frequently encountered and measurable metrics of an algorithm are:-

- speed or running time - the time it takes for an algorithm to complete, and
- 'space' - the memory or 'non-volatile storage' used by the algorithm during its operation. but also might apply to

- transmission size - such as required bandwidth during normal operation or
- size of external memory- such as temporary disk space used to accomplish its task and perhaps even
- the size of required 'longterm' disk space required after its operation to record its output or maintain its required function during its required useful lifetime (examples: a data table, archive or a computer log) and also
- the performance per watt and the total energy, consumed by the chosen hardware implementation (with its System requirements, necessary auxiliary support systems including interfaces, cabling, switching, cooling and security), during its required useful lifetime. See Total cost of ownership for other potential costs that might be associated with any particular implementation.

(An extreme example of these metrics might be to consider their values in relation to a repeated simple algorithm for calculating and storing (ð+n) to 50 decimal places running for say, 24 hours, either on a "pocket calculator" sized processor such as an ipod or an early mainframe operating in its own purpose-built heated or air conditioned unit.) The process of making code more efficient is known as optimization and in the case of automatic optimization (i.e. compiler optimization - performed by compilers on request or by default), usually focus on space at the cost of speed, or vice versa. There are also quite simple programming techniques and 'avoidance strategies' that can actually improve both at the same time, usually irrespective of hardware, software or language. Even the re-ordering of nested conditional statements - to put the least frequently occurring condition first (example: test patients for blood type ='AB-', before testing age > 18, since this type of blood occurs in only about 1 in 100 of the population - thereby eliminating the second test at runtime in 99% of instances), can reduce actual instruction path length, something an optimizing compiler would almost certainly not be aware of - but which a programmer can research relatively easily even without specialist medical knowledge.

HISTORY

The first machines that were capable of computation were severely limited by purely mechanical considerations. As later electronic machines were developed they were, in turn, limited by the speed of their electronic counterparts. As software replaced hard-wired circuits, the efficiency of algorithms also became important. It has long been recognized that the precise 'arrangement of processes' is critical in reducing elapse time.

- "In almost every computation a great variety of arrangements for the succession of the processes is possible, and various considerations must influence the selections amongst them for the purposes of a calculating engine. One essential object is to choose that arrangement which shall tend to reduce to a minimum the time necessary for completing the calculation"

Ada Lovelace 1815-1852, generally considered as 'the first programmer' who worked on Charles Babbage's early mechanical general-purpose computer

- "In established engineering disciplines a 12% improvement, easily obtained, is never considered marginal and I believe the same viewpoint should prevail in software engineering"

Extract from "Structured Programming with go to Statements" by Donald Knuth, renowned computer scientist and Professor Emeritus of the Art of Computer Programming at Stanford University.

- "The key to performance is elegance, not battalions of special cases" attributed to Jon Bentley and (Malcolm) Douglas McIlroy

SPEED

The absolute speed of an algorithm for a given input can simply be measured as the duration of execution (or clock time) and the results can be averaged over several executions to eliminate possible random effects. Most modern processors operate in a multi-processing & multi-programming environment so consideration must be made for parallel processes occurring on the same physical machine, eliminating these as far as possible. For superscalar processors, the speed of a given algorithm can sometimes be improved through instruction-level parallelism within a single processor (but, for optimal results, the algorithm may require some adaptation to this environment to gain significant advantage ('speedup'), becoming, in effect, an entirely different algorithm). A relative measure of an algorithms performance can sometimes be gained from the total instruction path length which can be determined by a run time Instruction Set Simulator (where available). An estimate of the speed of an algorithm can be determined in various ways. The most common method uses time complexity to determine the Big-O of an algorithm. See Run-time analysis for estimating how fast a particular algorithm may be according to its type (example: lookup unsorted list, lookup sorted list etc.) and in terms of scalability - its dependence on 'size of input', processor power and other factors.

MEMORY

Often, it is possible to make an algorithm faster at the expense of memory. This might be the case whenever the result of an 'expensive' calculation is cached rather than recalculating it afresh each time. The additional memory requirement would, in this case, be considered additional overhead although, in many situations, the stored result occupies very little extra space and can often be held in pre-compiled static storage, reducing not just processing time but also allocation & deallocation of working memory. This is a very common method of improving speed, so much so that some programming languages often add special features to support it, such as C++'s 'mutable' keyword. The memory requirement of an algorithm is actually two separate but related things:-

- The memory taken up by the compiled executable code (the object code or binary file) itself (on disk or equivalent, depending on the hardware and language).

This can often be reduced by preferring run-time decision making mechanisms (such as virtual functions and run-time type information) over certain compile-time decision making mechanisms (such as macro substitution and templates). This, however, comes at the cost of speed.

- Amount of temporary "dynamic memory" allocated during processing. For example, dynamically pre-caching results, as mentioned earlier, improves speed at the cost of this attribute. Even the depth of sub-routine calls can impact heavily on this cost and increase path length too, especially if there are 'heavy' dynamic memory requirements for the particular functions invoked. The use of copied function parameters (rather than simply using pointers to earlier, already defined, and sometimes static values) actually doubles the memory requirement for this particular memory metric (as well as carrying its own processing overhead for the copying itself. This can be particularly relevant for quite 'lengthy' parameters such as html script, JavaScript source programmes or extensive freeform text such as letters or emails.

REMATERIALIZATION

It has been argued that Rematerialization (re-calculating) may occasionally be more efficient than holding results in cache. This is the somewhat non-intuitive belief that it can be faster to re-calculate from the input - even if the answer is already known - when it can be shown, in some special cases, to decrease "register pressure". Some optimizing compilers have the ability to decide when this is considered worthwhile based on a number of criteria such as complexity and no side effects, and works by keeping track of the expression used to compute each variable, using the concept of available expressions.

This is most likely to be true when a calculation is very fast (such as addition or bitwise operations), while the amount of data which must be cached would be very large, resulting in inefficient storage. Small amounts of data can be stored very efficiently in registers or fast cache, while in most contemporary computers large amounts of data must be stored in slower memory or even slower hard drive storage, and thus the efficiency of storing data which can be computed quickly drops significantly.

PRECOMPUTATION

Precomputing a complete range of results prior to compiling, or at the beginning of an algorithm's execution, can often increase algorithmic efficiency substantially. This becomes advantageous when one or more inputs is constrained to a small enough range that the results can be stored in a reasonably sized block of memory. Because memory access is essentially constant in time complexity (except for caching delays), any algorithm with a component which has worse than constant efficiency over a small input range can be improved by precomputing values. In some cases efficient approximation algorithms can be obtained by computing a discrete subset of values and interpolating for intermediate input values, since interpolation is also a linear operation.

TRANSMISSION SIZE

Data compression algorithms can be useful because they help reduce the consumption of expensive resources, such as hard disk space or transmission bandwidth. This however also comes at a cost - which is additional processing time to compress and subsequently decompress. Depending upon the speed of the data transfer, compression may reduce overall response times which, ultimately, equates to speed - even though processing within the computer itself takes longer. For audio, MP3 is a compression method used extensively in portable sound systems. The efficiency of a data compression algorithm relates to the compression factor and speed of achieving both compression and decompression. For the purpose of archiving an extensive database, it might be considered worthwhile to achieve a very high compression ratio, since decompression is less likely to occur on the majority of the data.

DATA PRESENTATION

Output data can be presented to the end user in many ways - such as via punched tape or card, digital displays, local display monitors, remote computer monitors or printed. Each of these has its own inherent initial cost and, in some cases, an ongoing cost (e.g. refreshing an image from memory). As an example, the web site "Google" recently showed, as its logo, an image of the Vancouver olympics that is around 8K of gif image. The normally displayed Google image is a PNG image of 28K (or 48K), yet the raw text string for "Google" occupies only 6 octets or 48 bits (4,778 or 8192 *times* less). This graphically illustrates that how data is presented can significantly effect the overall efficiency of transmission (and also the complete algorithm - since both GIF and PNG images require yet more processing). It is estimated by "Internet World Stats" that there were 1,733,993,741 internet users in 2009 and, to transmit this new image to each one of them, would require around 136,000 *billion* (10^9)octets of data to be transmitted - at least once - into their personal web cache. In "Computational Energy Cost of TCP", co-authors Bokyung Wang and Suresh Singh examine the energy costs for TCP and calculated, for their chosen example, a cost of 0.022 Joules per packet (of approx 1489 octets). On this basis, a total of around 2,000,000,000 joules (2 GJ) of energy might be expended by TCP elements alone to display the new logo for all users for the first time. To maintain or re-display this image requires still more processing and consequential energy cost (in contrast to printed output for instance).

ENCODING AND DECODING METHODS (COMPARED AND CONTRASTED)

When data is encoded for any 'external' use, it is possible to do so in an almost unlimited variety of different formats that are sometimes conflicting. This content encoding (of the raw data) may be designed for:

- optimal readability – by humans
- optimal decoding speed – by other computer programmes
- optimal compression – for archiving or data transmission

- optimal compatibility – with “legacy” or other existing formats or programming languages
- optimal security – using encryption

  (For character level encoding, see the various encoding techniques such as EBCDIC or ASCII)

It is unlikely that all of these goals could be met with a single ‘generic’ encoding scheme and so a compromise will often be the desired goal and will often be compromised by the need for standardization and/or legacy and compatibility issues.

ENCODING EFFICIENTLY

For data encoding whose destination is to be input for further computer processing, readability is not an issue – since the receiving processors algorithm can decode the data to any other desirable form including human readable. From the perspective of algorithmic efficiency, minimizing subsequent decoding (with zero or minimal parsing) should take highest priority. The general rule of thumb is that any encoding system that ‘understands’ the underlying data structure - sufficiently to encode it in the first place - should be equally capable of easily encoding it in such a way that makes decoding it highly efficient. For variable length data with possibly omitted data values, for instance, this almost certainly means the utilization of declarative notation (i.e. including the length of the data item as a prefix to the data so that a de-limiter is not required and parsing completely eliminated). For keyword data items, tokenizing the key to an index (integer) after its first occurrence not only reduces subsequent data size but, furthermore, reduces future decoding overhead for the same items that are repeated. For more 'generic' encoding for efficient data compression see Arithmetic encoding and entropy encoding articles.

Historically, optimal encoding was not only worthwhile from an efficiency standpoint but was also common practise to conserve valuable memory, external storage and processor resources. Once validated a country name for example could be held as a shorter sequential country code which could then also act as an index for subsequent 'decoding', using this code as an entry number within a table or record number within a file. If the table or file contained fixed length entries, the code could easily be converted to an absolute memory address or disk address for fast retrieval. The ISBN system for identifying books is a good example of a practical encoding method which also contains a built-in hierarchy. According to recent articles in New Scientist and Scientific American; "TODAY'S telecommunications networks could use one ten-thousandth of the power they presently consume if smarter data-coding techniques were used", according to Bell Labs, based in Murray Hill, New Jersey It recognizes that this is only a theoretical limit but nevertheless sets itself a more realistic, practical goal of a 1,000 fold reduction within 5 years with future, as yet unidentified, technological changes.

EXAMPLES OF SEVERAL COMMON ENCODING METHODS

- Comma separated values (CSV - a list of data values separated by commas)
- Tab separated values (TSV) - a list of data values separated by 'tab' characters

- HyperText Markup Language (HTML) - the predominant markup language for web pages
- Extensible Markup Language (XML) - a generic framework for storing any amount of text or any data whose structure can be represented as a tree with at least one element - the root element.
- JavaScript Object Notation (JSON) - human-readable format for representing simple data structures

The last of these, (JSON) is apparently widely used for internet data transmission, primarily it seems because the data can be uploaded by a single JavaScript 'eval' statement - without the need to produce what otherwise would likely have been a more efficient purpose built encoder/decoder. There are in fact quite large amounts of repeated (and therefore redundant data) in this particular format, and also in HTML and XML source, that could quite easily be eliminated. XML is recognized as a verbose form of encoding. Binary XML has been put forward as one method of reducing transfer and processing times for XML and, while there are several competing formats, none has been widely adopted by a standards organization or accepted as a de facto standard. It has also been criticized by Jimmy Zhang for essentially trying to solve the wrong problem There are a number of freely available products on the market that partially compress HTML files and perform some or all of the following:

- merge lines
- remove unnecessary whitespace characters
- remove unnecessary quotation marks. For example, BORDER="0" will be replaced with BORDER=0)

- replace some tags with shorter ones (e.g. replace STRIKE tags with S, STRONG with B and EM with I)
- remove HTML comments (comments within scripts and styles are not removed)
- remove <!DOCTYPE..> tags
- remove named meta tags

The effect of this limited form of compression is to make the HTML code smaller and faster to load, but more difficult to read manually (so the original HTML code is usually retained for updating), but since it is predominantly meant to be processed only by a browser, this causes no problems. Despite these small improvements, HTML, which is the predominant language for the web still remains a predominantly *source* distributed, interpreted markup language, with high redundancy.

KOLMOGOROV COMPLEXITY

The study of encoding techniques has been examined in depth in an area of computer science characterized by a method known as Kolmogorov complexity where a value known as ('K') is accepted as 'not a computable function'. The Kolmogorov complexity of any computable object is the length of the shortest programme that computes it and then halts. The invariance theorem shows that it is not really important which computer is used. Essentially this implies that there is no automated method that can produce an optimum result and is therefore characterized by a requirement for human ingenuity or Innovation. See also Algorithmic probability.

EFFECT OF PROGRAMMING PARADIGMS

The effect that different programming paradigms have on algorithmic efficiency is fiercely contested, with both supporters and antagonists for each new paradigm. Strong supporters of structured programming, such as Dijkstra for instance, who favour entirely goto-less programmes are met with conflicting evidence that appears to nullify its supposed benefits. The truth is, even if the structured code itself contains no gotos, the optimizing compiler that creates the binary code almost certainly generates them (and not necessarily in the most efficient way). Similarly, OOP protagonists who claim their paradigm is superior are met with opposition from strong sceptics such as Alexander Stepanov who suggested that OOP provides a mathematically limited viewpoint and called it, "almost as much of a hoax as Artificial Intelligence" In the long term, benchmarks, using real-life examples, provide the only real hope of resolving such conflicts - at least in terms of run-time efficiency.

OPTIMIZATION TECHNIQUES

The word optimize is normally used in relation to an existing algorithm/computer programme (i.e. to improve upon completed code). In this section it is used both in the context of existing programmes and also in the design and implementation of new algorithms, thereby avoiding the most common performance pitfalls. It is clearly wasteful to produce a working programme - at first using an algorithm that ignores all efficiency issues - only to then have to redesign or rewrite sections of it if found to offer poor performance. Optimization can be broadly categorized into two domains:-

- Environment specific - that are essentially worthwhile only on certain platforms or particular computer languages
- General techniques - that apply irrespective of platform

ENVIRONMENT SPECIFIC

Optimization of algorithms frequently depends on the properties of the machine the algorithm will be executed on as well as the language the algorithm is written in and chosen data types. For example, a programmer might optimize code for time efficiency in an application for home computers (with sizable amounts of memory), but for code destined to be embedded in small, "memory-tight" devices, the programmer may have to accept that it will run more slowly, simply because of the restricted memory available for any potential software optimization. For a discussion of hardware performance, see article on Computer performance which covers such things as CPU clock speed, cycles per instruction and other relevant metrics. For a discussion on how the choice of particular instructions available on a specific machine effect efficiency, see later section 'Choice of instruction and data type'.

GENERAL TECHNIQUES

- Linear search such as unsorted table look-ups in particular can be very expensive in terms of execution time but can be reduced significantly through use of efficient techniques such as indexed arrays and binary searches. Using a simple linear search on first occurrence and using a cached result thereafter is an obvious compromise.

- Use of indexed programme branching, utilizing branch tables or "threaded code" to control programme flow, (rather than using multiple conditional IF statements or unoptimized CASE/SWITCH) can drastically reduce instruction path length, simultaneously reduce programme size and even also make a programme easier to read and more easily maintainable (in effect it becomes a 'decision table' rather than repetitive spaghetti code).
- Loop unrolling performed manually, or more usually by an optimizing compiler, can provide significant savings in some instances. By processing 'blocks' of several array elements at a time, individually addressed, (for example, within a While loop), much pointer arithmetic and end of loop testing can be eliminated, resulting in decreased instruction path lengths. Other Loop optimizations are also possible.

TUNNEL VISION

There are many techniques for improving algorithms, but focusing on a single favourite technique can lead to a "tunnel vision" mentality. For example, in this X86 assembly example, the author offers loop unrolling as a reasonable technique that provides some 40% improvements to his chosen example. However, the same example would benefit significantly from both inlining and use of a trivial hash function. If they were implemented, either as alternative or complementary techniques, an even greater percentage gain might be expected. A combination of optimizations may provide ever increasing speed, but selection of the most easily implemented and most effective technique, from a large repertoire of such techniques, is desirable as a starting point.

DEPENDENCY TREES AND SPREADSHEETS

Spreadsheets are a 'special case' of algorithms that self-optimize by virtue of their dependency trees that are inherent in the design of spreadsheets to reduce re-calculations when a cell changes. The results of earlier calculations are effectively cached within the workbook and only updated if another cells changed value effects it directly.

TABLE LOOKUP

Table lookups can make many algorithms more efficient, particularly when used to bypass computations with a high time complexity. However, if a wide input range is required, they can consume significant storage resources. In cases with a sparse valid input set, hash functions can be used to provide more efficient lookup access than a full table.

HASH FUNCTION ALGORITHMS

A hash function is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index to an array. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. Hash functions are frequently used to speed up table lookups. The choice of a hashing function (to avoid a linear or brute force search) depends critically on the nature of the input data, and their probability distribution in the intended application.

TRIVIAL HASH FUNCTION

Sometimes if the datum is small enough, a "trivial hash function" can be used to effectively provide constant time searches at almost zero cost. This is particularly relevant for single byte lookups (e.g. ASCII or EBCDIC characters)

SEARCHING STRINGS

Searching for particular text strings (for instance "tags" or keywords) in long sequences of characters potentially generates lengthy instruction paths. This includes searching for delimiters in comma separated files or similar processing which can be very simply and effectively eliminated (using declarative notation for instance). Several methods of reducing the cost for general searching have been examined and the "Boyer–Moore string search algorithm" (or Boyer–Moore–Horspool algorithm, a similar but modified version) is one solution that has been proven to give superior results to repetitive comparisons of the entire search string along the sequence.

HOT SPOT ANALYZERS

Special system software products known as "performance analyzers" are often available from suppliers to help diagnose "hot spots" - during actual execution of computer programmes - using real or test data - they perform a Performance analysis under generally repeatable conditions. They can pinpoint sections of the programme that might benefit from specifically targeted programmer optimization without necessarily spending time optimizing the rest of the code. Using programme re-runs, a measure of relative improvement can then be determined to decide if the optimization was successful and by what amount. Instruction Set Simulators can be used as an alternative to measure the instruction path length at the machine code level between selected execution paths, or on the entire execution. Regardless of the type of tool used, the quantitative values obtained can be used in combination with anticipated reductions (for the targeted code) to estimate a relative or absolute overall saving. For example if 50% of the total execution time (or path length) is absorbed in a subroutine whose speed can be doubled by programmer optimization, an overall saving of around 25% might be expected (Amdahl law). Efforts have been made at the University of California, Irvine to produce dynamic executable code using a combination of hot spot analysis and run-time programme trace tree. A JIT like dynamic compiler was built by Andreas Gal and others, "in which relevant (i.e., frequently executed) control flows are ...discovered lazily during execution"

BENCHMARKING & COMPETITIVE ALGORITHMS

For new versions of software or to provide comparisons with competitive systems, benchmarks are sometimes used which assist with gauging an algorithms relative performance. If a new sort algorithm is produced for example it can be compared with its predecessors to ensure that at least it is efficient as before with known data - taking into consideration any functional improvements. Benchmarks can be used by customers when comparing various products from alternative suppliers to estimate which product will best suit their specific requirements in terms of functionality and performance. For example in the mainframe world certain proprietary sort products from independent software companies such as Syncsort compete with products from the major suppliers such as IBM for speed. Some benchmarks provide opportunities for producing an analysis comparing the relative speed of various compiled and interpreted languages for example and *The Computer Language Benchmarks Game* compares the performance of implementations of typical programming problems in several programming languages. (Even creating "do it yourself" benchmarks to get at least some appreciation of the relative performance of different programming languages, using a variety of user specified criteria, is quite simple to produce as this "Nine language Performance roundup" by Christopher W. Cowell-Shah demonstrates by example)

COMPILED VERSUS INTERPRETED LANGUAGES

A compiled algorithm will, in general, execute faster than the equivalent interpreted algorithm simply because some processing is required even at run time to 'understand' (i.e. interpret) the instructions to effect an execution. A compiled programme will normally output an object or machine code equivalent of the algorithm that has already been processed by the compiler into a form more readily executed by microcode or the hardware directly. The popular Perl language is an example of an interpreted language and benchmarks indicate that it executes approximately 24 times more slowly than compiled C.

OPTIMIZING COMPILERS

Many compilers have features that attempt to optimize the code they generate, utilizing some of the techniques outlined in this study and others specific to the compilation itself. Loop optimization is often the focus of optimizing compilers because significant time is spent in programme loops and parallel processing opportunities can often be facilitated by automatic code re-structuring such as loop unrolling. Optimizing compilers are by no means perfect. There is no way that a compiler can guarantee that, for all programme source code, the fastest (or smallest) possible equivalent compiled programme is output; such a compiler is fundamentally impossible because it would solve the halting problem. Additionally, even optimizing compilers generally have no access to runtime metrics to enable them to improve optimization through 'learning'.

JUST-IN-TIME COMPILERS

'On-the-fly' processors known today as just-in-time or 'JIT' compilers combine features of interpreted languages with compiled languages and may also incorporate elements of optimization to a greater or lesser extent. Essentially the JIT compiler can compile small sections of source code statements (or bytecode) as they are newly encountered and (usually) retain the result for the next time the same source is processed. In addition, pre-compiled segments of code can be in-lined or called as dynamic functions that themselves perform equally fast as the equivalent 'custom' compiled function. Because the JIT processor also has access to run-time information (that a normal compiler can't have) it is also possible for it to optimize further executions depending upon the input and also perform other run-time introspective optimization as execution proceeds. A JIT processor may, or may not, incorporate self modifying code or its equivalent by creating 'fast path' routes through an algorithm. It may also use such techniques as dynamic Fractional cascading or any other similar runtime device based on collected actual runtime metrics. It is therefore entirely possible that a JIT compiler might (counter intuitively) execute even faster than an optimally 'optimized' compiled programme.

SELF-MODIFYING CODE

As mentioned above, just-in-time compilers often make extensive use of self-modifying code to generate the actual machine instructions required to be executed. The technique can also be used to reduce instruction path lengths in application programmes where otherwise repetitive conditional tests might otherwise be required within the main programme flow. This can be particularly useful where a sub routine may have embedded debugging code that is either active (testing mode) or inactive (production mode) depending upon some input parameter. A simple solution using a form of dynamic dispatching is where the sub routine entry point is dynamically 'swapped' at initialization, depending upon the input parameter. Entry point A) includes the debugging code prologue and entry point B) excludes the prologue; thus eliminating all overhead except the initial 'test and swap' (even when test/debugging is selected, when the overhead is simply the test/debugging code itself).

GENETIC ALGORITHM

In the world of performance related algorithms it is worth mentioning the role of genetic algorithms which compete using similar methods to the natural world in eliminating inferior algorithms in favour of more efficient versions.

OBJECT CODE OPTIMIZERS

Some proprietary programme optimizers such as the "COBOL Optimizer" developed by Capex Corporation in the mid 1970's for COBOL, actually took the unusual step of optimizing the Object code (or binary file) after normal compilation. This type of optimizer, recently sometimes referred to as a "post pass" optimizer or peephole optimizer, depended, in this case, upon knowledge of 'weaknesses' in the standard IBM COBOL compiler and actually replaced (or patched) sections of the object code with more efficient code. A number of other suppliers have recently adopted the same approach.

ALIGNMENT OF DATA

Most processors execute faster if certain data values are aligned on word, doubleword or page boundaries. If possible design/specify structures to satisfy appropriate alignments. This avoids exceptions.

LOCALITY OF REFERENCE

To reduce Cache miss exceptions by providing good spatial locality of reference, specify 'high frequency'/volative working storage data within defined structure(s) so that they are also allocated from contiguous sections of memory (rather than possibly scattered over many pages). Group closely related data values also 'physically' close together within these structures. Consider the possibility of creating an 'artificial' structure to group some otherwise unrelated, but nevertheless frequently referenced, items together.

CHOICE OF INSTRUCTION OR DATA TYPE

Particularly in an Assembly language (although also applicable to HLL statements), the choice of a particular 'instruction' or data type, can have a large impact on execution efficiency. In general, instructions that process variables such as signed or unsigned 16-bit or 32-bit integers are faster than those that process floating point or packed decimal. Modern processors are even capable of executing multiple 'fixed point' instructions in parallel with the simultaneous execution of a floating point instruction. If the largest integer to be encountered can be accommodated by the 'faster' data type, defining the variables as that type will result in faster execution - since even a non-optimizing compiler will, in-effect, be 'forced' to choose appropriate instructions that will execute faster than would have been the case with data types associated with 'slower' instructions. Assembler programmers (and optimizing compiler writers) can then also benefit from the ability to perform certain common types of arithmetic for instance - division by 2, 4, 8 etc. by performing the very much faster binary shift right operations (in this case by 1, 2 or 3 bits). If the choice of input data type is not under the control of the programmer, although prior conversion (outside of a loop for instance) to a faster data type carries some overhead, it can often be worthwhile if the variable is then to be used as a loop counter, especially if the count could be quite a high value or there are many input values to process. As mentioned above, choice of individual assembler instructions (or even sometimes just their order of execution) on particular machines can effect the efficiency of an algorithm. See Assembly Optimization Tips for one quite numerous arcane list of various technical (and sometimes non-intuitive) considerations for choice of assembly instructions on different processors that also discusses the merits of each case. Sometimes microcode or hardware quirks can result in unexpected performance differences between processors that assembler programmers can actively code for - or else specifically avoid if penalties result - something even the best optimizing compiler may not be designed to handle.

DATA GRANULARITY

The greater the granularity of data definitions (such as splitting a geographic address into separate street/city/ postal code fields) can have performance overhead implications during processing. Higher granularity in this example implies more procedure calls in Object-oriented programming and parallel computing environments since the additional objects are accessed via multiple method calls rather than perhaps one.

SUBROUTINE GRANULARITY

For structured programming and procedural programming in general, great emphasis is placed on designing programmes as a hierarchy of (or at least a set of) subroutines. For object oriented programming, the method call (a subroutine call) is the standard method of testing and setting all values in objects and so increasing data granularity consequently causes increased use of subroutines. The greater the granularity of subroutine usage, the larger the proportion of processing time devoted to the mechanism of the subroutine linkages themselves.The presence of a (called) subroutine in a programme contributes nothing extra to the functionality of the programme. The extent to which subroutines (and their consequent memory requirements) influences the overall performance of the complete algorithm depends to a large extent on the actual implementation. In assembly language programmes, the invocation of a subroutine need not involve much overhead, typically adding just a couple of machine instructions to the normal instruction path length, each one altering the control flow either *to* the subroutine or returning *from* it (saving the state on a stack being optional, depending on the complexity of the subroutine and its requirement to reuse general purpose registers). In many cases, small subroutines that perform frequently used data transformations using 'general purpose' work areas can be accomplished without the need to save or restore any registers, including the return register.

By contrast, HLL programmes typically always invoke a 'standard' procedure call (the *calling convention*), which involves saving the programme state by default and usually allocating additional memory on the stack to save all registers and other relevant state data (the prologue and epilogue code). Recursion in a HLL programme can consequently consume significant overhead in both memory and execution time managing the stack to the required depth. Guy Steele pointed out in a 1977 paper that a *well-designed* programming language implementation *can* have very low overheads for procedural abstraction (but laments, in most implementations, that they seldom achieve this in practice - being "rather thoughtless or careless in this regard"). Steele concludes that "we should have a healthy respect for procedure calls" (because they are powerful) but he also cautioned "use them sparingly" See section Avoiding costs for discussion of how inlining subroutines can be used to improve performance. For the Java language, use of the "final" keyword can be used to force method inlining (resulting in elimination of the method call, no dynamic dispatch and the possibility to constant-fold the value - with no code executed at runtime)

CHOICE OF LANGUAGE / MIXED LANGUAGES

Some computer languages can execute algorithms more efficiently than others. As stated already, interpreted languages often perform less efficiently than compiled languages. In addition, where possible, 'high-use', and time-dependent sections of code may be written in a language such as assembler that can usually execute faster and make better use of resources on a particular platform than the equivalent HLL code on the same platform. This section of code can either be statically called or dynamically invoked (external function) or embedded within the higher level code (e.g. Assembler instructions embedded in a 'C' language program). The effect of higher levels of abstraction when using a HLL has been described as the *Abstraction penalty* Programmers who are familiar with assembler language (in addition to their chosen HLL) and are also familiar with the code that will be generated by the HLL, under known conditions, can sometimes use HLL language primitives of that language to generate code almost identical to assembler language. This is most likely to be possible only in languages that support pointers such as PL/1 or C. This is facilitated if the chosen HLL compiler provides an optional assembler listing as part of its printout so that various alternatives can be explored without also needing specialist knowledge of the compiler internals.

SOFTWARE VALIDATION VERSUS HARDWARE VALIDATION

An optimization technique that was frequently taken advantage of on 'legacy' platforms was that of allowing the hardware (or microcode) to perform validation on numeric data fields such as those coded in (or converted to) packed decimal (or packed BCD). The choice was to either spend processing time checking each field for a valid numeric content in the particular internal representation chosen or simply assume the data was correct and let the hardware detect the error upon execution. The choice was highly significant because to check for validity on multiple fields (for sometimes many millions of input records), it could occupy valuable computer resources. Since input data fields were in any case frequently built from the output of earlier computer processing, the actual probability of a field containing invalid data was exceedingly low and usually the result of some ‘corruption’. The solution was to incorporate an ‘event handler’ for the hardware detected condition (‘data exception)’ that would intercept the occasional errant data field and either ‘report, correct and continue’ or, more usually, abort the run with a core dump to try to determine the reason for the bad data.

Similar event handlers are frequently utilized in today’s web based applications to handle other exceptional conditions but repeatedly parsing data input, to ensure its validity before execution, has nevertheless become much more commonplace - partly because processors have become faster (and the perceived need for efficiency in this area less significant) but, predominantly - because data structures have become less ‘formalized’ (e.g. .csv and .tsv files) or uniquely identifiable (e.g. packed decimal). The potential savings using this type of technique may have therefore fallen into general dis-use as a consequence and therefore repeated data validations and repeated data conversions have become an accepted overhead. Ironically, one consequence of this move to less formalized data structures is that a corruption of say, a numeric binary integer value, will not be detected at all by the hardware upon execution (for instance: is an ASCII hexadecimal value ‘20202020’ a valid signed or unsigned binary value - or simply a string of blanks that has corrupted it?)

ALGORITHMS FOR VECTOR & SUPERSCALAR PROCESSORS

Algorithms for vector processors are usually different than those for scalar processors since they can process multiple instructions and/or multiple data elements in parallel. The process of converting an algorithm from a scalar to a vector process is known as vectorization and methods for automatically performing this transformation as efficiently as possible are constantly sought. There are intrinsic limitations for implementing instruction level parallelism in Superscalar processors but, in essence, the overhead in deciding for certain if particular instruction sequences can be processed in parallel can sometimes exceed the efficiency gain in so doing. The achievable reduction is governed primarily by the (somewhat obvious) law known as Amdahl's law, that essentially states that the improvement from parallel processing is determined by its slowest sequential component. Algorithms designed for this class of processor therefore require more care if they are not to unwittingly disrupt the potential gains.

AVOIDING COSTS

- Defining variables as integers for indexed arrays instead of floating point will result in faster execution.
- Defining structures whose structure length is a multiple of a power of 2 (2,4,8,16 etc.), will allow the compiler to calculate array indexes by shifting a binary index by 1, 2 or more bits to the left, instead of using a multiply instruction will result in faster execution. Adding an otherwise redundant short filler

variable to ‘pad out’ the length of a structure element to say 8 bytes when otherwise it would have been 6 or 7 bytes may reduce overall processing time by a worthwhile amount for very large arrays. See for generated code differences for C as for example.

- Storage defined in terms of bits, when bytes would suffice, may inadvertently involve extremely long path lengths involving bitwise operations instead of more efficient single instruction ‘multiple byte’ copy instructions. (This does not apply to ‘genuine’ intentional bitwise operations - used for example instead of multiplication or division by powers of 2 or for TRUE/FALSE flags.)
- Unnecessary use of allocated dynamic storage when static storage would suffice, can increase the processing overhead substantially - both increasing memory requirements and the associated allocation/ deallocation path length overheads for each function call.
- Excessive use of function calls for very simple functions, rather than in-line statements, can also add substantially to instruction path lengths and stack/unstack overheads. For particularly time critical systems that are not also code size sensitive, automatic or manual inline expansion can reduce path length by eliminating all the instructions that call the function and return from it. (A conceptually similar method, loop unrolling, eliminates the instructions required to set up and terminate a loop by, instead; repeating the instructions inside the

loop multiple times. This of course eliminates the branch back instruction but may also increase the size of the binary file or, in the case of JIT built code, dynamic memory. Also, care must be taken with this method, that re-calculating addresses for each statement within an unwound indexed loop is not more expensive than incrementing pointers within the former loop would have been. If absolute indexes are used in the generated (or manually created) unwound code, rather than variables, the code created may actually be able to avoid generated pointer arithmetic instructions altogether, using offsets instead).

MEMORY MANAGEMENT

Whenever memory is *automatically* allocated (for example in HLL programmes, when calling a procedure or when issuing a system call), it is normally released (or 'freed'/ 'deallocated'/ 'deleted') automatically when it is no longer required - thus allowing it to be re-used for another purpose *immediately*. Some memory management can easily be accomplished by the compiler, as in this example. However, when memory is *explicitly* allocated (for example in OOP when "new" is specified for an object), releasing the memory is often left to an asynchronous 'garbage collector' which does not necessarily release the memory at the earliest opportunity (as well as consuming some additional CPU resources deciding if it can be). The current trend nevertheless appears to be towards taking full advantage of this fully automated method, despite the tradeoff in efficiency - because it is claimed that it makes programming easier. Some functional languages are known as 'lazy functional languages' because of the significant use of garbage collection and can consume much more memory as a result.

- Array processing may simplify programming but use of separate statements to sum different elements of the same array(s) may produce code that is not easily optimized and that requires multiple passes of the arrays that might otherwise have been processed in a single pass. It may also duplicate data if array slicing is used, leading to increased memory usage and copying overhead.
- In OOP, if an object is known to be immutable, it can be copied simply by making a copy of a reference to it instead of copying the entire object. Because a reference (typically only the size of a pointer) is usually much smaller than the object itself, this results in memory savings and a boost in execution speed.

READABILITY, TRADE OFFS AND TRENDS

One must be careful, in the pursuit of good coding style, not to over-emphasize efficiency. Frequently, a clean, readable and 'usable' design is much more important than a fast, efficient design that is hard to understand. There are exceptions to this 'rule' (such as embedded systems, where space is tight, and processing power minimal) but these are rarer than one might expect. However, increasingly, for many 'time critical' applications such as air line reservation systems, point-of-sale applications, ATMs (cashpoint machines), Airline Guidance systems, Collision avoidance systems and numerous modern web based applications - operating in a real-time environment where speed of response is fundamental - there is little alternative.

DETERMINING IF OPTIMIZATION IS WORTHWHILE

The essential criteria for using optimized code are of course dependent upon the expected use of the algorithm. If it is a new algorithm and is going to be in use for many years and speed is relevant, it is worth spending some time designing the code to be as efficient as possible from the outset. If an existing algorithm is proving to be too slow or memory is becoming an issue, clearly something must be done to improve it. For the average application, or for one-off applications, avoiding inefficient coding techniques and encouraging the compiler to optimize where possible may be sufficient. One simple way (at least for mathematicians) to determine whether an optimization is worthwhile is as follows: Let the original time and space requirements (generally in Big-O notation) of the algorithm be O_1 and O_2. Let the new code require N_1 and N_2 time and space respectively. If $N_1N_2 < O_1O_2$, the optimization should be carried out. However, as mentioned above, this may not always be true.

IMPLICATIONS FOR ALGORITHMIC EFFICIENCY

A recent report, published in December 2007, from Global Action Plan, a UK-based environmental organization found that computer servers are "at least as great a threat to the climate as SUVs or the global aviation industry" drawing attention to the carbon footprint of the IT industry in the UK. According to an Environmental Research Letters report published in September 2008, "Total power used by information technology equipment in data centers represented about 0.5% of world electricity consumption in 2005. When cooling and auxiliary infrastructure are included, that figure is about 1%. The total data center power demand in 2005 is equivalent (in capacity terms) to about seventeen 1000 MW power plants for the world." Some media reports claim that performing two Google searches from a desktop computer can generate about the same amount of carbon dioxide as boiling a kettle for a cup of tea, according to new research; however, the factual accuracy of this comparison is disputed, and the author of the study in question asserts that the two-searches-tea-kettle statistic is a misreading of his work.

Greentouch, a recently established consortium of leading Information and Communications Technology (ICT) industry, academic and non-governmental research experts, has set itself the mission of reducing reduce energy consumption per user by a factor of 1000 from current levels. "A thousand-fold reduction is roughly equivalent to being able to power the world's communications networks, including the Internet, for three years using the same amount of energy that it currently takes to run them for a single day". The first meeting in February 2010 will establish the organization's five-year plan, first year deliverables and member roles and responsibilities. Intellectual property issues will be addressed and defined in the forum's initial planning meetings. The conditions for research and the results of that research will be high priority for discussion in the initial phase of the research forum's development. Computers having become increasingly more powerful over the past few decades, emphasis was on a 'brute force' mentality. This may have to be reconsidered in the light of these reports and more effort placed in future on reducing carbon footprints through optimization. It is a timely reminder that algorithmic efficiency is just another aspect of the more general thermodynamic efficiency. The genuine economic benefits of an optimized algorithm are, in any case, that more processing can be done for the same cost or that useful results can be shown in a more timely manner and ultimately, acted upon sooner.

CRITICISM OF THE CURRENT STATE OF PROGRAMMING

- David May FRS a British computer scientist and currently Professor of Computer Science at University of Bristol and founder and CTO of XMOS Semiconductor, believes one of the problems is that there is a reliance on Moore's law to solve inefficiencies. He has advanced an 'alternative' to Moore's law (May's law) stated as follows:

Software efficiency halves every 18 months, compensating Moore's Law. In ubiquitous systems, halving the instructions executed can double the battery life and big data sets bring big opportunities for better software and algorithms: Reducing the number of operations from N x N to N x log(N) has a dramatic effect when N is large...

for N = 30 billion, this change is as good as 50 years of technology improvements

- Software author Adam N. Rosenburg in his blog "*The failure of the Digital computer*", has described the current state of programming as nearing the "Software event horizon", (alluding to the fictitious "*shoe event horizon*" described by Douglas Adams in his *Hitchhiker's Guide to the Galaxy* book). He estimates there has been a 70 dB factor loss of productivity or "99.99999 percent, of its ability to deliver the goods", since the 1980s - "When Arthur C. Clarke compared the reality of computing in 2001 to the computer HAL in his book 2001: A Space Odyssey, he pointed out how wonderfully small and powerful computers were but how disappointing computer programming had become".
- Conrad Weisert gives examples, some of which were published in ACM SIGPLAN (Special Interest Group on Programming Languages) Notices, December, 1995 in: "Atrocious Programming Thrives"

5

Sorting of Algorithms

INTRODUCTION

It is not always possible to say that one algorithm is better than another, as relative performance can vary depending on the type of data being sorted. In some situations, most of the data are in the correct order, with only a few items needing to be sorted. In other situations the data are completely mixed up in a random order and in others the data will tend to be in reverse order. Different algorithms will perform differently according to the data being sorted. Four common algorithms are the exchange or bubble sort, the selection sort, the insertion sort and the quick sort. The selection sort is a good one to use with students. It is intuitive and very simple to Programme. It offers quite good performance, its particular strength being the small number of exchanges needed. For a given number of data items, the selection sort always goes through a set number of comparisons and exchanges, so its performance is predictable. Procedure SelectionSort (d: DataArrayType; n: integer) {n is the number of elements}

```
for k = 1 to n-1 do begin
small = k
for j = k+1 to n do
if d[j] < d[small] then small = j
{Swap elements k and small}
Swap(d, k, small)
end
```

EXCHANGE (BUBBLE) SORT

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9
1st pass	27	1	63	64	58	14	9	72
2nd pass	1	27	63	58	14	9	64	72
3rd pass	1	27	58	14	9	63	64	72...

The first two data items (27 and 63) are compared and the smaller one placed on the left hand side. The second and third items (63 and 1) are then compared and the smaller one placed on the left and so on.

After all the data has been passed through once, the largest data item (72) will have "bubbled" through to the end of the list. At the end of the second pass, the second largest data item (64) will be in the second last position. For n data items, the process continues for n-1 passes, or until no exchanges are made in a single pass.

INSERTION SORT

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9
1st pass	27	63	1	72	64	58	9	14
2nd pass	27	63	1	72	64	9	14	58
3rd pass	27	63	1	72	9	14	58	64...

The insertion sort starts with the last two elements and creates a correctly sorted sub-list, which in the example contains 9 and 14. It then looks at the next element (58) and inserts it into the sub-list in its correct position.

It takes the next element (64) and does the same, continuing until the sub-list contains all the data.

SELECTION SORT

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9
1st pass	1	63	27	72	64	58	14	9
2nd pass	1	9	27	72	64	58	14	63
3rd pass	1	9	14	72	64	58	27	63...

The selection sort marks the first element (27). It then goes through the remaining data to find the smallest number (1). It swaps this with the first element and the smallest element is now in its correct position. It then marks the second element (63) and looks through the remaining data for the next smallest number (9). These two numbers are then swapped. This process continues until n-1 passes have been made.

QUICK SORT

Element	1	2	3	4	5	6	7	8
Data	27	63	1	72	64	58	14	9
1st pass	1	9	63	72	64	58	14	27
2nd pass	1	9	14	27	64	58	72	63
3rd pass	1	9	14	27	58	63	72	64
4th pass	1	9	14	27	58	63	64	72 sorted!

The quick sort takes the last element (9) and places it such that all the numbers in the left sub-list are smaller and all the numbers in the right sub-list are bigger. It then quick sorts the left sub-list ({1}) and then quick sorts the right sub-list. This is a recursive algorithm, since it is defined in terms of itself. This reduces the complexity of Programmeming it, however it is the least intuitive of the four algorithms.

COMPARING THE ALGORITHMS

There are two important factors when measuring the performance of a sorting algorithm. The algorithms have to compare the magnitude of different elements and they have to move the different elements around. So counting the number of comparisons and the number of exchanges or moves made by an algorithm offer useful performance measures. When sorting large record structures, the number of exchanges made may be the principal performance criterion, since exchanging two records will involve a lot of work. When sorting a simple array of integers, then the number of comparisons will be more important.

It has been said that the only thing going for the bubble (exchange) sort is its catchy name. The logic of the algorithm is simple to understand and it is fairly easy to Programme. It can also be Programmemed to detect when it has finished sorting. The selection sort, by comparison, always goes through the same amount of work regardless of the data and the quick sort performs particularly badly with ordered data. However, in general the bubble sort is a very inefficient algorithm.

The insertion sort is a little better and whilst it cannot detect that it has finished sorting, the logic of the algorithm means that it comes to a rapid conclusion when dealing with sorted data.

The selection sort is a good one to use with students. It is intuitive and very simple to Programme. It offers quite good performance, its particular strength being the small number of exchanges needed. For a given number of data items, the selection sort always goes through a set number of comparisons and exchanges, so its performance is predictable.

The first three algorithms all offer O(n2) performance, that is sorting times increase with the square of the number of elements being sorted. That means that if you double the number of elements being sorted, then there will be a four-fold increase in the time taken.

Ten times more elements increases the time taken by a factor of 100! This is not a problem with small data sets, but with hundreds or thousands of elements, this becomes very significant. With most large data sets, the quick sort is a vastly superior algorithm (although as you might expect, it is much more complex), as the table below shows.

RANDOM DATA SET: NUMBER

Sort/Elements	50	100	200	300	400	500
Selection Sort	1225	4950	19900	44850	79800	124750
Exchange Sort	1410	5335	20300	45650	79866	126585
Insertion Sort	1391	5399	20473	44449	78779	123715
Quick Sort	399	990	1954	3384	5066	6256

It should be pointed out that the methods above all belong to one family, they are all internal sorting algorithms. This means that they can only be used when the entire data structure to be sorted can be held in the computer's main memory. There will be situations where this is not possible, for example when sorting a very large transaction file which is stored on, say, magnetic tape or disc.

DESCRIPTION

BUBBLE SORT

Exchange two adjacent elements if they are out of order. Repeat until array is sorted.

This is a slow algorithm:

```
  #include <stdlib.h>
  #include <stdio.h>
  #define uint32 unsigned int
  typedef int (*CMPFUN)(int, int);
  void ArraySort(int This[], CMPFUN fun_ptr,
uint32 ub)
  {
  /* bubble sort */
   uint32 indx;
   uint32 indx2;
   int temp;
   int temp2;
   int flipped;
   if (ub ⇐ 1)
   return;
   indx = 1;
   do
   {
    flipped = 0;
    for (indx2 = ub – 1; indx2 ⇒ indx; —
  indx2)
   {
   temp = This[indx2];
   temp2 = This[indx2 – 1];
   if ((*fun_ptr)(temp2, temp) > 0)
    {
      This[indx2 – 1] = temp;
      This[indx2] = temp2;
      flipped = 1;
      }
    }
   } while ((++indx < ub) && flipped);
  }
```

```
 #define ARRAY_SIZE 14
 int my_array[ARRAY_SIZE];
 void fill_array()
 {
  int indx;
  for (indx=0; indx < ARRAY_SIZE; ++indx)
  {
  my_array[indx] = rand();
  }
  /* my_array[ARRAY_SIZE - 1] = ARRAY_SIZE/
3; */
  }
 int cmpfun(int a, int b)
 {
  if (a > b)
   return 1;
  else if (a < b)
   return -1;
  else
   return 0;
  }
 int main()
  {
     int indx;
     int indx2;
     for (indx2 = 0; indx2 < 80000; ++indx2)
     {
      fill_array();
      ArraySort(my_array, cmpfun, ARRAY_SIZE);
      for (indx=1; indx < ARRAY_SIZE; ++indx)
      {
       if (my_array[indx - 1] > my_array[indx])
        {
            printf("bad sort\n");
            return(1);
            }
          }
        }
        return(0);
    }
```

SELECTION SORT

Find the largest element in the array, and put it in the proper place. Repeat until array is sorted. This is also slow.

```
   #include <stdlib.h>
   #include <stdio.h>
   #define uint32 unsigned int
   typedef int (*CMPFUN)(int, int);
   void ArraySort(int This[], CMPFUN fun_ptr,
uint32 the_len)
  {
     /* selection sort */
   uint32 indx;
   uint32 indx2;
   uint32 large_pos;
   int temp;
   int large;
   if (the_len <= 1)
   return;
   for (indx = the_len - 1; indx > 0; --indx)
   {
  /* find the largest number, then put it at
the end of the array */
  large = This[0];
  large_pos = 0;
  for (indx2 = 1; indx2 <= indx; ++indx2)
 {
  temp = This[indx2];
  if ((*fun_ptr)(temp,large) > 0)
  {
    large = temp;
    large_pos = indx2;
    }
   }
  This[large_pos] = This[indx];
  This[indx] = large;
  }
 }
 #define ARRAY_SIZE 14
 int my_array[ARRAY_SIZE];
 void fill_array()
 {
  int indx;
  for (indx=0; indx < ARRAY_SIZE; ++indx)
 {
  my_array[indx] = rand();
  }
  /* my_array[ARRAY_SIZE - 1] = ARRAY_SIZE/
3; */
  }
  int cmpfun(int a, int b)
  {
   if (a > b)
    return 1;
   else if (a < b)
    return -1;
   else
    return 0;
  }
  int main()
  {
     int indx;
     int indx2;
     for (indx2 = 0; indx2 < 80000; ++indx2)
     {
        fill_array();
        ArraySort(my_array, cmpfun, ARRAY_SIZE);
        for (indx=1; indx < ARRAY_SIZE; ++indx)
        {
          if (my_array[indx - 1] > my_array[indx])
          {
           printf("bad sort\n");
           return(1);
          }
        }
     }
  return(0);
 }
```

INSERTION SORT

Scan successive elements for out of order item, then insert the item in the proper place. Sort small array fast, big array very slowly.

```
 #include <stdlib.h>
 #include <stdio.h>
 #define uint32 unsigned int
 typedef int (*CMPFUN)(int, int);
 void ArraySort(int This[], CMPFUN fun_ptr,
uint32 the_len)
{
 /* insertion sort */
 uint32 indx;
```

```
  int cur_val;
  int prev_val;
 if (the_len Ü 1)
   return;
 prev_val = This[0];
 for (indx = 1; indx < the_len; ++indx)
 {
  cur_val = This[indx];
  if ((*fun_ptr)(prev_val, cur_val) > 0)
  {
  /* out of order: array[indx-1] > array[indx]
*/
  uint32 indx2;
  This[indx] = prev_val;/* move up the larger
item first */
  /* find the insertion point for the smaller
item */
  for (indx2 = indx - 1; indx2 > 0;)
  {
    int temp_val = This[indx2 - 1];
    if ((*fun_ptr)(temp_val, cur_val) > 0)
    {
      This[indx2—] = temp_val;
      /* still out of order, move up 1 slot to
     make room */
     }
     else
      break;
  }
  This[indx2] = cur_val;/* insert the smaller
item right here */
 }
 else
 {
   /* in order, advance to next element */
   prev_val = cur_val;
   }
  }
 }

 #define ARRAY_SIZE 14

 int my_array[ARRAY_SIZE];

 uint32 fill_array()
```

```
{
int indx;
uint32 checksum = 0;
for (indx=0; indx < ARRAY_SIZE; ++indx)
{
  checksum += my_array[indx] = rand();
 }
 return checksum;
 }
 int cmpfun(int a, int b)
 {
   if (a > b)
     return 1;
   else if (a < b)
    return -1;
   else
     return 0;
 }
 int main()
 {
   int indx;
   int indx2;
   uint32 checksum1;
   uint32 checksum2;
   for (indx2 = 0; indx2 < 80000; ++indx2)
   {
     checksum1 = fill_array();
     ArraySort(my_array, cmpfun, ARRAY_SIZE);
     for (indx=1; indx < ARRAY_SIZE; ++indx)
     {
       if (my_array[indx - 1] > my_array[indx])
       {
        printf("bad sort\n");
        return(1);
      }
   }
   checksum2 = 0;
   for (indx=0; indx < ARRAY_SIZE; ++indx)
   {
  checksum2 += my_array[indx];
 }
  if (checksum1 != checksum2)
  {
    printf("bad checksum%d%d\n", checksum1,
  checksum2);
  }
}
```

```
  return(0);
 }
```

QUICK SORT

Partition array into two segments. The first segment all elements are less than or equal to the pivot value. The second segment all elements are greater or equal to the pivot value. Sort the two segments recursively. Quicksort is fastest on average, but sometimes unbalanced partitions can lead to very slow sorting.

```
   #include <stdlib.h>
   #include <stdio.h>
   #define INSERTION_SORT_BOUND 16/* boundary
 point to use insertion sort */
   #define uint32 unsigned int
   typedef int (*CMPFUN)(int, int);
   /* explain function
   * Description:
   * fixarray::Qsort() is an internal subroutine
  that implements quick sort.
   *
   * Return Value: none
   */
  void Qsort(int This[], CMPFUN fun_ptr, uint32
 first, uint32 last)
  {
   uint32 stack_pointer = 0;
   int first_stack[32];
   int last_stack[32];
   for (;;)
  {
      if (last - first ⇐ INSERTION_SORT_BOUND)
      {
         /* for small sort, use insertion sort
      */
         uint32 indx;
         int prev_val = This[first];
         int cur_val;
         for (indx = first + 1; indx Ü last;
      ++indx)
        {
         cur_val = This[indx];
        if ((*fun_ptr)(prev_val, cur_val) > 0)
   {
    /* out of order: array[indx-1] > array[indx]
   */
    uint32 indx2;
   This[indx] = prev_val;/* move up the larger
item first */
   /* find the insertion point for the smaller
  item */
   for (indx2 = indx - 1; indx2 > first;)
   {
     int temp_val = This[indx2 - 1];
     if ((*fun_ptr)(temp_val, cur_val) > 0)
     {
       This[indx2—] = temp_val;
       /* still out of order, move up 1 slot
     to make room */
     }
     else
       break;
    }
    This[indx2] = cur_val;/* insert the smaller
   item right here */
  }
  else
  {
     /* in order, advance to next element */
      prev_val = cur_val;
    }
   }
}
else
{
   int pivot;
  /* try quick sort */
   {
    int temp;
    uint32 med = (first + last) >> 1;
    /* Choose pivot from first, last, and median
   position. */
    /* Sort the three elements. */
    temp = This[first];
    if ((*fun_ptr)(temp, This[last]) > 0)
    {
      This[first] = This[last]; This[last] =
     temp;
    }
     temp = This[med];
     if ((*fun_ptr)(This[first], temp) > 0)
    {
      This[med] = This[first]; This[first] =
    temp;
    }
    temp = This[last];
    if ((*fun_ptr)(This[med], temp) > 0)
    {
      This[last] = This[med]; This[med] = temp;
    }
     pivot = This[med];
 }
 {
    uint32 up;
    {
      uint32 down;
        /* First and last element will be loop
     stopper. */
        /* Split array into two partitions. */
        down = first;
        up = last;
        for (;;)
        {
          do
          {
            ++down;
 } while ((*fun_ptr)(pivot, This[down]) > 0);
  do
  {
    —up;
 } while ((*fun_ptr)(This[up], pivot) > 0);
  if (up > down)
  {
     int temp;
    /* interchange L[down] and L[up] */
     temp = This[down]; This[down]= This[up];
This[up] = temp;
 }
  else
  break;
 }
 }
```

```
  {
    uint32 len1;/* length of first segment */
    uint32 len2;/* length of second segment
  */
    len1 = up - first + 1;
    len2 = last - up;
    /* stack the partition that is larger */
    if (len1 Þ len2)
    {
      first_stack[stack_pointer] = first;
      last_stack[stack_pointer++] = up;
      first = up + 1;
       /* tail recursion elimination of
       * Qsort(This,fun_ptr,up + 1,last)
       */
     }
     else
    {
      first_stack[stack_pointer] = up + 1;
      last_stack[stack_pointer++] = last;
     last = up;
      /* tail recursion elimination of
      * Qsort(This,fun_ptr,first,up)
      */
     }
   }
    continue;
 }
 /* end of quick sort */
 }
 if (stack_pointer > 0)
 {
   /* Sort segment from stack. */
   first = first_stack[—stack_pointer];
   last = last_stack[stack_pointer];
  }
  else
    break;
  }/* end for */
 }
 void ArraySort(int This[], CMPFUN fun_ptr,
uint32 the_len)
 {
  Qsort(This, fun_ptr, 0, the_len - 1);
 }
```

```
#define ARRAY_SIZE 250000
int my_array[ARRAY_SIZE];
uint32 fill_array()
{
 int indx;
 uint32 checksum = 0;
 for (indx=0; indx < ARRAY_SIZE; ++indx)
 {
   checksum += my_array[indx] = rand();
 }
 return checksum;
}
int cmpfun(int a, int b)
{
 if (a > b)
  return 1;
 else if (a < b)
  return –1;
else
 return 0;
}
int main()
{
  int indx;
  uint32 checksum1;
  uint32 checksum2 = 0;
  checksum1 = fill_array();
  ArraySort(my_array, cmpfun, ARRAY_SIZE);
  for (indx=1; indx < ARRAY_SIZE; ++indx)
  {
     if (my_array[indx – 1] > my_array[indx])
      {
        printf("bad sort\n");
        return(1);
       }
  }
  for (indx=0; indx < ARRAY_SIZE; ++indx)
  {
  checksum2 += my_array[indx];
 }
 if (checksum1 != checksum2)
 {
   printf("bad checksum%d%d\n", checksum1,
 checksum2);
   return(1);
  }
   return(0);
 }
```

MERGE SORT

Start from two sorted runs of length 1, merge into a single run of twice the length. Repeat until a single sorted run is left. Mergesort needs N/2 extra buffer. Performance is second place on average, with quite good speed on nearly sorted array. Mergesort is stable in that two elements that are equally ranked in the array will not have their relative positions flipped.

```
   #include <stdlib.h>
   #include <stdio.h>
   #define uint32 unsigned int
  typedef int (*CMPFUN)(int, int);
  #define INSERTION_SORT_BOUND 8/* boundary
point to use insertion sort */
  void ArraySort(int This[], CMPFUN fun_ptr,
uint32 the_len)
  {
  uint32 span;
  uint32 lb;
  uint32 ub;
  uint32 indx;
  uint32 indx2;
  if (the_len Ü 1)
   return;
  span = INSERTION_SORT_BOUND;
  /* insertion sort the first pass */
  {
    int prev_val;
    int cur_val;
    int temp_val;
    for (lb = 0; lb < the_len; lb += span)
    {
      if ((ub = lb + span) > the_len) ub =
    the_len;
      prev_val = This[lb];
     for (indx = lb + 1; indx < ub; ++indx)
     {
        cur_val = This[indx];
        if ((*fun_ptr)(prev_val, cur_val) > 0)
        {
          /* out of order: array[indx-1] >
        array[indx] */
          This[indx] = prev_val;/* move up the
         larger item first */
          /* find the insertion point for the
        smaller item */
         for (indx2 = indx - 1; indx2 > lb;)
       {
         temp_val = This[indx2 - 1];
          if ((*fun_ptr)(temp_val, cur_val) >
         0)
          {
            This[indx2—] = temp_val;
            /* still out of order, move up 1
          slot to make room */
        }
       else
        break;
       }
      This[indx2] = cur_val;/* insert the
    smaller item right here */
      }
      else
      {
     /* in order, advance to next element */
     prev_val = cur_val;
    }
   }
  }
 }
  /* second pass merge sort */
  {
   uint32 median;
   int* aux;
   aux = (int*) malloc(sizeof(int) * the_len/
 2);

  while (span < the_len)
  {
    /* median is the start of second file */
    for (median = span; median < the_len;)
    {
```

```
  indx2 = median - 1;
  if ((*fun_ptr)(This[indx2], This[median])
> 0)
 {
   /* the two files are not yet sorted */
  if ((ub = median + span) > the_len)
  {
    ub = the_len;
  }
  /* skip over the already sorted largest
elements */
   while ((*fun_ptr)(This[—ub], This[indx2])
⇒ 0)
{
}
/* copy second file into buffer */
for (indx = 0; indx2 < ub; ++indx)
{
  *(aux + indx) = This[++indx2];
}
 —indx;
indx2 = median - 1;
lb = median - span;
/* merge two files into one */
 for (;;)
{
  if ((*fun_ptr)(*(aux + indx), This[indx2])
⇒ 0)
  {
    This[ub—] = *(aux + indx);
    if (indx > 0) —indx;
     else
     {
     /* second file exhausted */
      for (;;)
      {
        This[ub—] = This[indx2];
        if (indx2 > lb) —indx2;
        else goto mydone;/* done */
      }
    }
 }
 else
 {
   This[ub—] = This[indx2];
```

```
        if (indx2 > lb) —indx2;
        else
        {
           /* first file exhausted */
           for (;;)
           {
             This[ub—] = *(aux + indx);
             if (indx > 0) —indx;
             else goto mydone;/* done */
            }
           }
        }
     }
  }
  mydone:
  median += span + span;
  }
   span += span;
  }
    free(aux);
   }
  }
  #define ARRAY_SIZE 250000
  int my_array[ARRAY_SIZE];
  uint32 fill_array()
  {
    int indx;
    uint32 sum = 0;
    for (indx=0; indx < ARRAY_SIZE; ++indx)
    {
       sum += my_array[indx] = rand();
    }
    return sum;
   }

   int cmpfun(int a, int b)
   {
     if (a > b)
        return 1;
    else if (a < b)
     return -1;
   else
     return 0;
   }
   int main()
  {
  int indx;
  uint32 checksum, checksum2;
  checksum = fill_array();
  ArraySort(my_array, cmpfun, ARRAY_SIZE);
  checksum2 = my_array[0];
  for (indx=1; indx < ARRAY_SIZE; ++indx)
  {
    checksum2 += my_array[indx];
    if (my_array[indx - 1] > my_array[indx])
    {
      printf("bad sort\n");
       return(1);
     }
  }
  if (checksum != checksum2)
  {
      printf("bad checksum%d%d\n", checksum,
    checksum2);
       return(1);
   }
   return(0);
   }
```

HEAP SORT

Form a tree with parent of the tree being larger than its children. Remove the parent from the tree successively. On average, Heapsort is third place in speed. Heapsort does not need extra buffer, and performance is not sensitive to initial distributions.

```
  #include <stdlib.h>
  #include <stdio.h>
  #define uint32 unsigned int
  typedef int (*CMPFUN)(int, int);
  void ArraySort(int This[], CMPFUN fun_ptr,
uint32 the_len)
  {
    /* heap sort */
    uint32 half;
    uint32 parent;
    if (the_len ⇐ 1)
      return;
    half = the_len >> 1;
    for (parent = half; parent Þ 1; —parent)
  {
    int temp;
    int level = 0;
    uint32 child;
    child = parent;
    /* bottom-up downheap */
    /* leaf-search for largest child path */
    while (child ⇐ half)
    {
      ++level;
      child += child;
      if ((child < the_len) &&
      ((*fun_ptr)(This[child], This[child - 1])
      > 0))
     ++child;
   }
   /* bottom-up-search for rotation point */
   temp = This[parent - 1];
   for (;;)
  {
    if (parent == child)
      break;
    if ((*fun_ptr)(temp, This[child - 1]) ⇐
  0)
       break;
    child >⇒ 1;
    —level;
  }
  /* rotate nodes from parent to rotation point
 */
  for (;level > 0; —level)
  {
    This[(child >> level) - 1] =
       This[(child >> (level - 1)) - 1];
   }
   This[child - 1] = temp;
 }
  —the_len;
 do
 {
   int temp;
   int level = 0;
   uint32 child;
  /* move max element to back of array */
  temp = This[the_len];
  This[the_len] = This[0];
  This[0] = temp;
  child = parent = 1;
  half = the_len >> 1;
  /* bottom-up downheap */
  /* leaf-search for largest child path */
  while (child ⇐ half)
  {
    ++level;
    child += child;
    if ((child < the_len) &&
       ((*fun_ptr)(This[child], This[child - 1])
       > 0))
  ++child;
}
/* bottom-up-search for rotation point */
for (;;)
{
 if (parent == child)
   break;
 if ((*fun_ptr)(temp, This[child - 1]) ⇐ 0)
   break;
 child >⇒ 1;
  —level;
 }
 /* rotate nodes from parent to rotation point
*/
 for (;level > 0; —level)
 {
 This[(child >> level) - 1] =
   This[(child >> (level - 1)) - 1];
 }
 This[child - 1] = temp;
 } while (—the_len ⇒ 1);
}
#define ARRAY_SIZE 250000
int my_array[ARRAY_SIZE];
void fill_array()
{
  int indx;
  for (indx=0; indx < ARRAY_SIZE; ++indx)
  {
   my_array[indx] = rand();
 }
}

int cmpfun(int a, int b)
{
  if (a > b)
    return 1;
 else if (a < b)
    return -1;
 else
    return 0;
 }
 int main()
{
int indx;
fill_array();
ArraySort(my_array, cmpfun, ARRAY_SIZE);
for (indx=1; indx < ARRAY_SIZE; ++indx)
{
  if (my_array[indx - 1] > my_array[indx])
  {
   printf("bad sort\n");
   return(1);
  }
 }
 return(0);
}
```

SHELL SORT

Sort every Nth element in an array using insertion sort. Repeat using smaller N values, until N = 1. On average, Shellsort is fourth place in speed. Shellsort may sort some distributions slowly.

```
  #include <stdlib.h>
  #include <stdio.h>
  #define uint32 unsigned int
  typedef int (*CMPFUN)(int, int);
   /* Calculated from the combinations of 9 *
 (4^n - 2^n) + 1,
  * and 4^n - 3 * 2^n + 1
  */
  uint32 hop_array[] =
  {
```

```
  1,
  5,
  19,
  41,
  109,
  209,
  505,
  929,
  2161,
  3905,
  8929,
  16001,
  36289,
  64769,
  146305,
  260609,
  587521,
  1045505,
  2354689,
  4188161,
  9427969,
  16764929,
  37730305,
  67084289,
  150958081,
  268386305,
  603906049,
  1073643521,
  2415771649,
  0xffffffff};
  void ArraySort(int This[], CMPFUN fun_ptr,
uint32 the_len)
  {
    /* shell sort */
    int level;
    for (level = 0; the_len > hop_array[level];
   ++level);
    do
    {
        uint32 dist;
        uint32 indx;
        dist = hop_array[—level];
        for (indx = dist; indx < the_len; ++indx)
        {
          int cur_val;
          uint32 indx2;
          cur_val = This[indx];
          indx2 = indx;
          do
          {
            int early_val;
            early_val = This[indx2 - dist];
            if ((*fun_ptr)(early_val, cur_val)
⇐             0)
              break;
           This[indx2] = early_val;
            indx2 -= dist;
           } while (indx2 ⇒ dist);
           This[indx2] = cur_val;
        }
      } while (level Þ 1);
    }
    #define ARRAY_SIZE 250000
    int my_array[ARRAY_SIZE];
    uint32 fill_array()
    {
      int indx;
      uint32 checksum = 0;
      for (indx=0; indx < ARRAY_SIZE; ++indx)
      {
        checksum += my_array[indx] = rand();
      }
      return checksum;
   }
   int cmpfun(int a, int b)
  {
    if (a > b)
       return 1;
    else if (a < b)
    return -1;
   else
    return 0;
  }
  int main()
  {
  int indx;
  uint32 sum1;
  uint32 sum2;
  sum1 = fill_array();
  ArraySort(my_array, cmpfun, ARRAY_SIZE);
  for (indx=1; indx < ARRAY_SIZE; ++indx)
  {
    if (my_array[indx - 1] > my_array[indx])
  {
    printf("bad sort\n");
    return(1);
   }
 }
 for (indx = 0; indx < ARRAY_SIZE; ++indx)
 {
   sum2 += my_array[indx];
 }
if (sum1 != sum2)
 {
   printf("bad checksum\n");
   return(1);
  }
  return(0);
 }
```

COMBO SORT

Sorting algorithms can be mixed and matched to yield the desired properties. We want fast average performance, good worst case performance, and no large extra storage requirement. We can achieve the goal by starting with the Quicksort (fastest on average). We modify Quicksort by sorting small partitions by using Insertion Sort (best with small partition). If we detect two partitions are badly balanced, we sort the larger partition by Heapsort (good worst case performance). Of course we cannot undo the bad partitions, but we can stop the possible degenerate case from continuing to generate bad partitions.

```
  #include <stdlib.h>
  #include <stdio.h>
  #define uint32 unsigned int
  typedef int (*CMPFUN)(int, int);
  void HelperHeapSort(int This[], CMPFUN
fun_ptr, uint32 first, uint32 the_len)
  {
    /* heap sort */
    uint32 half;
    uint32 parent;
    if (the_len Ü 1)
        return;
    half = the_len >> 1;
    for (parent = half; parent ⇒ 1; —parent)
    {
       int temp;
       int level = 0;
       uint32 child;
       child = parent;
        /* bottom-up downheap */
       /* leaf-search for largest child path */
       while (child ⇐ half)
   {
    ++level;
    child += child;
       if ((child < the_len) &&
            ((*fun_ptr)(This[first + child],
          This[first + child - 1]) > 0))
         ++child;
  }
  /* bottom-up-search for rotation point */
  temp = This[first + parent - 1];
  for (;;)
  {
    if (parent == child)
        break;
     if ((*fun_ptr)(temp, This[first + child -
    1]) ⇐ 0)
        break;
    child >⇒ 1;
    —level;
  }
  /* rotate nodes from parent to rotation point
*/
  for (;level > 0; —level)
  {
    This[first + (child >> level) - 1] =
       This[first + (child >> (level - 1)) -
    1];
   }
    This[first + child - 1] = temp;
  }
```

```
 —the_len;
 do
 {
   int temp;
   int level = 0;
   uint32 child;
   /* move max element to back of array */
   temp = This[first + the_len];
   This[first + the_len] = This[first];
   This[first] = temp;
   child = parent = 1;
   half = the_len >> 1;
   /* bottom-up downheap */
  /* leaf-search for largest child path */
  while (child ⇐ half)
  {
  ++level;
  child += child;
  if ((child < the_len) &&
       ((*fun_ptr)(This[first + child],
  This[first + child - 1]) > 0))
      ++child;
 }
 /* bottom-up-search for rotation point */
 for (;;)
 {
   if (parent == child)
       break;
    if ((*fun_ptr)(temp, This[first + child -
   1]) ⇐ 0)
    break;
   child >⇒ 1;
    —level;
 }
 /* rotate nodes from parent to rotation point
*/
 for (;level > 0; —level)
 {
  This[first + (child >> level) - 1] =
    This[first + (child >> (level - 1)) - 1];
 }
 This[first + child - 1] = temp;
 } while (—the_len Þ 1);
}
#define INSERTION_SORT_BOUND 16/* boundary          point
```

```
to use insertion sort */
 /* explain function
 * Description:
 * fixarray::Qsort() is an internal subroutine      t h a t
implements quick sort.
 *
 * Return Value: none
 */
 void Qsort(int This[], CMPFUN fun_ptr, uint32
first, uint32 last)
 {
   uint32 stack_pointer = 0;
   int first_stack[32];
   int last_stack[32];
   for (;;)
 {
   if (last - first Ü INSERTION_SORT_BOUND)
   {
      /* for small sort, use insertion sort */
      uint32 indx;
      int prev_val = This[first];
      int cur_val;
      for (indx = first + 1; indx ⇐ last;
   ++indx)
  {
   cur_val = This[indx];
   if ((*fun_ptr)(prev_val, cur_val) > 0)
 {
   uint32 indx2;
   /* out of order */
   This[indx] = prev_val;
   for (indx2 = indx - 1; indx2 > first; —
indx2)
  {
    int temp_val = This[indx2 - 1];
    if ((*fun_ptr)(temp_val, cur_val) > 0)
  {
    This[indx2] = temp_val;
  }
  else
    break;
  }
    This[indx2] = cur_val;
  }
   else
```

```
{
  /* in order, advance to next element */
  prev_val = cur_val;
  }
 }
}
 else
 {
   int pivot;
  /* try quick sort */
{
  int temp;
  uint32 med = (first + last) >> 1;
  /* Choose pivot from first, last, and median
 position. */
  /* Sort the three elements. */
  temp = This[first];
  if ((*fun_ptr)(temp, This[last]) > 0)
{
  This[first] = This[last]; This[last] = temp;
}
temp = This[med];
if ((*fun_ptr)(This[first], temp) > 0)
{
  This[med] = This[first]; This[first] = temp;
}
temp = This[last];
if ((*fun_ptr)(This[med], temp) > 0)
{
 This[last] = This[med]; This[med] = temp;
}
 pivot = This[med];
}
{
  uint32 up;
  {
    uint32 down;
    /* First and last element will be loop
  stopper. */
   /* Split array into two partitions. */
   down = first;
   up = last;
   for (;;)
 {
    do
 {
     ++down;
   } while ((*fun_ptr)(pivot, This[down]) >
0);

 do
   {
      —up;
   } while ((*fun_ptr)(This[up], pivot) >
0);

    if (up > down)
    {
      int temp;
      /* interchange L[down] and L[up] */
       temp = This[down]; This[down]=
      This[up]; This[up] = temp;
     }
      else
        break;
     }
}
     {
       uint32 len1;/* length of first
     segment */
       uint32 len2;/* length of second
     segment */
       len1 = up - first + 1;
       len2 = last - up;
       if (len1 ⇒ len2)
       {
          if ((len1 >> 5) > len2)
           {
          /* badly balanced partitions, heap
        sort first segment */
           HelperHeapSort(This, fun_ptr,
        first,len1);
      }
      else
      {
          first_stack[stack_pointer] = first;
          /* stack first segment */
           last_stack[stack_pointer++] = up;
        }
        first = up + 1;
```

```
                /* tail recursion elimination of
                 * Qsort(This,fun_ptr,up + 1,last)
                 */
            }
             else
            {
              if ((len2 >> 5) > len1)
              {
                 /* badly balanced partitions, heap
                sort second segment */
                 HelperHeapSort(This, fun_ptr, up
                + 1, len2);
            }
            else
            {
              first_stack[stack_pointer] = up +1;
            /* stack second segment */
              last_stack[stack_pointer++] = last;
           }
            last = up;
             /* tail recursion elimination of
              * Qsort(This,fun_ptr,first,up)
              */
           }
         }
      continue;
    }
    /* end of quick sort */
   }
   if (stack_pointer > 0)
   {
      /* Sort segment from stack. */
      first = first_stack[—stack_pointer];
      last = last_stack[stack_pointer];
   }
   else
      break;
   }/* end for */
  }
  void ArraySort(int This[], CMPFUN fun_ptr,
uint32 the_len)
  {
   Qsort(This, fun_ptr, 0, the_len - 1);
  }
  #define ARRAY_SIZE 250000
```

```
int my_array[ARRAY_SIZE];
void fill_array()
{
int indx;
for (indx=0; indx < ARRAY_SIZE; ++indx)
{
  my_array[indx] = rand();
 }
}
int cmpfun(int a, int b)
{
  if (a > b)
     return 1;
  else if (a < b)
     return -1;
  else
     return 0;
}
int main()
{
 int indx;
 fill_array();
 ArraySort(my_array, cmpfun, ARRAY_SIZE);
 for (indx=1; indx < ARRAY_SIZE; ++indx)
 {
   if (my_array[indx - 1] > my_array[indx])
   {
      printf("bad sort\n");
      return(1);
   }
 }
  return(0);
 }
```