# Introduction to computer science

Warren D. Smith
wds@research.NJ.NEC.COM

date = September 23, 1998

*Abstract* — **Everything you need to know about Computer Science in 1 lecture? Computer scientists will already know most of what's in here. If they want to learn something they probably don't already know, I recommend reading Bennett's paper on "logical depth."**

*Keywords* —

Contents



## 1 Computer science speak

"$f(n) = O(g(n))$" means: there exists a constant $c > 0$ so that $f(n) < cg(n)$ for all sufficiently large $n$.

Sometimes computer scientists use the word "polynomial" to mean "$O(n^\alpha)$ for some real constant $\alpha > 0$" and the word "exponential" to mean "$\exp O(n^\alpha)$ for some constant $\alpha > 0$." But other times they just mean the usual meanings, e.g. "$O(n^\alpha)$ for some integer constant $\alpha > 0$" and "$\exp(cn)$ for some constant $c > 0$."

## 2 Models of computation, including Turing machine, RAM

### 2.1 Turing machine

The "Turing machine" is a simple abstract computational model consisting of a "head" and a semi-infinite "tape." The tape consists of "squares," each labeled with one character from a finite alphabet $\Sigma$. The head can be in any one of a finite set $S$ of states. Initially the head is at the left end of the tape (at square 1) and is in the "start" state. The operation of the Turing machine is totally determined by whatever is initially written on the tape – which usually would be a finite "program" string and a finite "input" string, followed by an infinite sequence of "blank" characters – and the "transition function" of the machine.

The transition function maps

$$\Sigma_{\rm old} \times S_{\rm old} \to \left\{ \begin{matrix} \text{go left,} \\ \text{go right,} \\ \text{stand still} \end{matrix} \right\} \times \Sigma_{\rm new} \times S_{\rm new}, \qquad (1)$$

that is, depending on the current state, the head moves 1 step left, right, or stays still; the current tape square is overwritten by a new character, and the head state changes to a new state, and the transition function says which.

As computation proceeds, the Turing machine can write "output" and "intermediate results" on the tape. When it feels "done" it can enter a special "halt" state, or decide

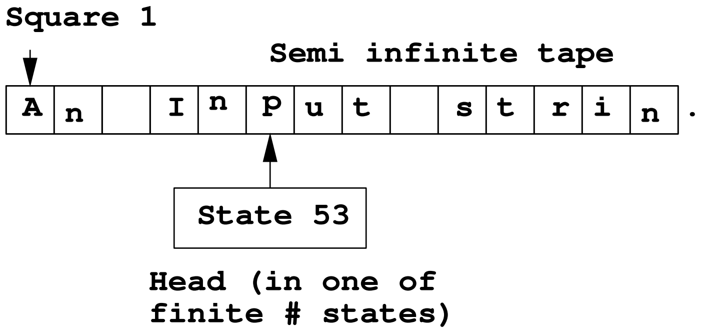


Figure 1: Turing machine.

to fall off the left end of the tape (same effect). The tape serves as an unboundedly large "memory."

Turing believed that this model's capabilities encapsulated what anybody could reasonably call "computation."

Imagine a mathematician with a finite brain (hence it's capable of being in a finite set of states). He can crawl over an infinite sheet of paper. Based on what he sees written there and what is in his brain, he can erase and write new stuff. He can crawl at finite speed; he has to write with characters in a fixed-size font. Turing argued that it would be ridiculous for the mathematician's brain to have an infinite number of states (if so, two of them would be infinitesimally "close together" and hence indistinguishable) and similarly ridiculous if the alphabet had an infinite number of characters. Hence this model is "clearly" the best one could hope for. And this is the same as the Turing machine, except that the Turing machine only has a 1-dimensional "sheet of paper" – which, we'll see later, does not matter.

Many Turing machines are boring. For example, you could devise a transition function which makes it just sit there and do nothing. Or which makes it move right forever.

A "universal" Turing machine is a machine with a sufficiently useful transition function that it may be "programmed" (with a finite length program pre-written on the tape), to emulate any other Turing machine, i.e. with any other transition function. (The program will include a description of that other machine's transition function table.) The key amazing observation, by Turing, is that *universality is possible.* Universal Turing machines have been explicitly written down. In fact there is a an ongoing contest [12] to find ultrasimple ones. For example, universality may now be achieved with an alphabet of only 2 characters – '0' and '1' and a head with only 24 states. Or, if you use a 6-character alphabet, you only need a head with 4 states.

Once you have universality, you can essentially write any computer program and demand the Turing machine execute it. If you don't like programming in Turing, you could write a BASIC interpreter and thenceforth write a BASIC program on the Turing machine's tape. (In fact, the existence of interpreters for languages could be thought of as a proof of the existence of universal TMs.)

"Church's thesis" is the vague all-embracing claim that any "reasonable model of computation" is equivalent (perhaps with some slowdown function) to a universal Turing machine. The "strong thesis" is that any $T(N)$-step computation on an $N$-bit long input on some other kind of machine, may be emulated by a Turing machine with polynomial($T(N), N$) runtime, and with a program only a constant additive amount longer.

The "constant additive amount" is the constant length "emulator" program.

People proved Church's thesis for a large number of reasonable and not-so-reasonable models of computation.

**Example: many, multidimensional tapes.** Suppose you have a Turing machine with 79 (or any constant number of) tapes, not just one, and which can take a step (move $\pm 1$ or 0) on all 79 tapes simultaneously, and can read and write all 79 characters at once. Suppose each "tape" is not a 1-dimensional, but 65-dimensional (any finite dimensional), grid. This can be emulated by a plain Turing machine with polynomial slowdown.

**Example: RAM with tapes.** Suppose the machine comes equipped with an infinite "RAM" (random access memory) allowing read/write access to any element of an infinite table in 1 step, assuming the binary "address" had been pre-written onto some tape. Each entry of the table is itself an infinite tape, which may be read 1 character at a time. Again, although this might seem to allow superpolynomially more power (nonlocal access patterns; move exponentially far in polynomial number of steps) an emulation algorithm has been devised with only polynomial slowdown.

**Example: Cellular automaton.** Suppose we have a "cellular automaton" consisting of an infinite $d$-dimensional grid, $d$ constant. Each grid square has a character written in it from some finite alphabet. Each time step, each grid square decides to change state according to some fixed "transition function" mapping the character, and the states of all characters within some constant distance away, into the new character. Initially, everything is blank except for a *finite* subset of the grid. This model might seem much more powerful since it has "infinitely parallel processing"... but again an emulation algorithm has been devised featuring only polynomial slowdown.

**Sample Church proof sketch:** Just so we can say we did one... here's a way to emulate a 2-tape Turing machine with a 1-tape Turing machine. The squares numbered 1 mod 4 will be simulated tape #1, the squares 2 mod 4 will be simulated tape #2. The squares 0 mod 4 will consist of 2-state markers saying "if you want to get to the current location on tape 2, you have to move further (right, left)." Ditto for the squares 3 mod 4 for tape 1. The head will have enough states that it can remember where it is mod 4, and can remember what tape (1 or 2) it is looking for at the moment, and can remember the last few tape squares the simulated 2-tape machine read. To emulate 1 step of the 2-tape machine requires a few walks back and forth, but in time $T(N)$ the 2-tape machine could not have moved its 2 heads more than $T(N)$ apart, so the walks take $O(T(N))$ steps. Hence we emulate a $T$-step computation on the 2-tape machine by a $T^2$-step computation on the

1-tape machine – polynomial slowdown. QED.

### 2.2 Random Turing machines

**Theorem 1** *Fix an alphabet. Let the cardinality of the set of head states for a Turing machine be $H$. Define a Turing machine's transition function uniformly at random subject to the constraint that a head state must transition to a head state within a constant distance from it (mod $H$). Then: As $H \to \infty$, the probability tends to 1 that the reuslting Turing machine will be universal.*

**Proof sketch.** The probability is a small, but nonzero, positive constant $c$ that the Turing machine transition function, restricted to any particular 4-wide subinterval mod $H$ of head-state space, happens to be identical to the transition function for the known 6-character 4-state UTM. The probability that this keeps on not happening is $\leq (1-c)^{H/4} \to 0$. QED.

**Remarks.** This also works if the "distance metric" among the head states is based on a grid in any fixed dimensional space. The point of this is that one can imagine plausible physical systems, arising by natural random processes, with probability $\approx 1$ of being universal computers.

### 2.3 Real RAM

The "real RAM" model is rather like FORTRAN. We have a finite read-only memory containing the program. We also have an infinite read/write memory. Each word of memory contains a real number. In one step you can perform an arithmetic operation on a real number, test if a number is $<$, $>$ or $=$ to 0, read or write a real number to or from memory or to or from the output, and branch anywhere in the program. This model is very convenient for writing programs (writing programs for a Turing machine is very painful in comparison). But this model is usually regarded as unrealistically powerful. For example, Shamir showed [see [3] ex. 4.5.4 #38] that a real RAM equipped with the "floor" function ($\lfloor x \rfloor$ rounds to greatest integer below $x$) can factor $N$-digit integers into primes in only polynomial($N$) steps. (You can emulate floor if you have cosine and arccosine and $\pi$, so only allowing commonplace analytic functions does not get you out of this.) Shamir's idea involved "enormously parallel processing" – each real arithmetic operation is really doing an infinite amount of work, and Shamir figured out a way to abuse that power. If, however, we make a deal never to try to unrealistically abuse the power of the real RAM – for example, we figure out how many decimal places of precision we "really"' would need to carry... and don't allow algorithms that involve too much precision... this is OK.

### 2.4 Finite wordsize RAM

"RAM" means "random access model." This is just like the real RAM above, except each word of memory is only $w$ bits wide for some finite $w$. This model is most similar to present day computers.

### 2.5 Counter machines

In the other direction, we can try to make our model of computation even simpler than a Turing machine. One of the simplest computers with universal power is the "counter machine." This consists of 3 "counters" containing non-negative integers. In 1 step, each counter can be incremented or decremented by 1 (or left as is). And there is a "head." The head is in one of a finite number of states. The head can test each counter to see if it is $> 0$ or $= 0$ (decrements have no effect on 0), then, based on these tests and the old head state, a transition function tells us the new head state to go to and what to do to the counters. There is a universal counter machine which can emulate any Turing machine – albeit with exponential (not polynomial) slowdown. The idea of the proof is: one counter contains $2^n$ where the Turing machine tape position is $n$. Another counter is used as scratch space to count to $2^n$. The third counter contains the Turing machine tape, and altering bit $n$ of that tape is accomplished by adding or subtracting $2^n$ from it.

See [11] chapter 14 for a demonstration that only *two* counters suffice. In fact, a machine with only 2 non-negative integer "counters" and a head which runs a finite fixed program made of only 2 kinds instructions (increment counter $j$; decrement and test for zero counter $j$) suffices for universal computation.

### 2.6 FRACTRAN

Related to this is Conway's "FRACTRAN." This machine has a single non-negative integer, plus a head. In one step, we can multiply or divide the integer by 2 or 3. We can also test the integer to see if dividing by 2 or 3 would have resulted in a noninteger fraction, i.e. see if the integer is zero or not mod 2 or 3. A transition function, based on these tests and the old head state, gives the new head state and the amount to multiply or divide the integer by.

**Proof:** FRACTRAN is equivalent to Minsky's 2-counter machine – the counters are the number of factors of 2 and 3 in the integer.

### 2.7 Other peripherals

We could grant our Turing machine other interesting capabilities. For example, it could have access to a random bit generator (equivalently: access to a tape on which is pre-written an infinite stream of random bits).

Or we could add "input" and "output" bits and allow Turing machines to talk to one another through wires connected to these bits.

In such cases the transition function depends also on the input bits and its value tells what to write on the output bits.

## 3 What is an "Algorithm?"

Suppose we have a universal Turing machine with 4 tapes. The first (read only) tape is the "program." The second (read only) tape is the "input." The third (read/write) tape is memory, and the fourth tape (write only) is the "output."

An "algorithm" is a finite length string on the program tape of our UTM, with the property that, *no matter what* is written on the input tape, the UTM will always *in finite time* enter an "I'm done" or "halted" state. At this point: you can examine the output tape.

Algorithms are a subset of "programs." Programs need not halt. Algorithms must halt. For example, an infinite loop is a program, but not an algorithm.

Even better than an algorithm, would be a "provable algorithm," i.e. one which comes with a proof that it always will eventually halt. Even better than that would be an algorithm which comes with a proof that, for an $N$-bit input, it will always halt within time $T(N)$, with some explicit and easy to compute formula $T(N)$. Even better than that, would be if $T(N)$ is slow growing.

## 4 Undecidability, uncomputability, unprovability

A "problem class" is: a map from input strings to output strings. I.e. given some input string, the problem solver is supposed to compute an output string with specified properties. A "problem instance" is a problem class plus a specified input string. A "yes-no" problem has a 1-bit output.

**Example problem:** Integer squaring. (Input: the string representing a decimal integer. Output: string representing its square.)

**Example yes-no problem:** produce as output 1 bit, indicating if input integer is or isn't prime.

Warning: don't get confused between problem classes (e.g. INTEGER SQUARING) and problem instances (please square 15). Often one abuses notation and just calls a problem class or instance a "problem." leading the unwary into confusion.

If an algorithm exists to solve all problems of your class, that problem class is "decidable." If no algorithm exists, the problem class is "undecidable" (or some people call this "Turing undecidable").

### *4.1 Turing undecidability*

Here is a result which terrified Hilbert when Turing discovered it. But nowadays it is considered trivial.

**Theorem 2** *Undecidable problems exist.*

**Proof.** (A.Turing) Consider the following problem class, known as the **"halting problem."**

INPUT: The input string $I$ is the description of a Turing machine transition function and the nonblank initial portion of its tape.

OUTPUT: A bit: 1 if that TM would have halted on that input, 0 otherwise.

The halting problem is undecidable. To see that, suppose the contrary – some algorithm $A$ exists to solve the halting problem. In that case, let me modify the code for algorithm $A$, getting modified algorithm $A'$ (actually $A'$ is a program, not an algorithm, since it need not halt) as follows:

1. $A'$ inputs only the Turing machine description $D$, and just uses a copy of $D$ in place of the nonblank tape portion.
2. If $A$ outputs 1, instead $A'$ goes into an infinite loop; if $A$ outputs 0, $A'$ immediately halts.

It then follows that $A'$ would respond to its own description on tape as its own input, by halting if and only if $A'$ did not halt with description($A'$) as input, and not halting if and only if $A'$ would halt! We have reached a contradiction. QED[1].

A different way to look at this argument is as a "Cantor diagonalization argument." For that view, define the "computable Boolean sequences" to be those (finite or infinite) bit strings which a Universal Turing Machine equipped with an output device for outputting bits, could output, assuming finite input. Obviously the computable Boolean sequences are countable. Hence almost all Boolean sequences (i.e., real numbers in $[0, 1]$, in binary form) are uncomputable. A specific example of an uncomputable number is as follows. List the computable Boolean sequences in order. Consider the "diagonal" sequence $B_1, B_2,...$ defined by "$B_n$ is the complement of the $n$th bit in the $n$th Boolean sequence." To prove this sequence is not computable, assume the contrary. Then it must agree with the $n$th computable squence for some $n$; but it does not since it disaagrees in bit $n$. QED.

**Warning:** There is no such thing as a *particular* undecidable instance of the halting problem. That is because the following algorithm decides your favorite instance: "print 'It halts'." (Or not.) The whole problem *class* is undecidable in the sense that no decision algorithm exists which always gets the right answer for all instances.

Here are four other problems known to be Turing-undecidable: the "tiling problem," "Collatz $3x+1$-like problems," "the triviality problem for finitely presented groups," and the "diophantine equations problem." (All of these problems are in fact, in some sense, at least as hard as any Turing-undecidable problem, in the sense that any problem may be phrased as an instance of one of them.)

**Tiling problem** [Wang,Berger; see R.M.Robinson: Invent.Math. 12 (1971) 177-209 for cleaner formulations].

INPUT: The descriptions of a finite set $F$ of plane tiles – simple polygons drawn on the square grid using only up, down, right or left motions forming a self-avoiding closed path. These include one distinguished "starting tile."

OUTPUT: 1 bit, indicating whether a tiling of the plane exists using an infinite number of tiles, each selected from $F$, and with the special "starting tile" being at the origin.

**Proof sketch:** Use tiles which are rectangles with little wiggly bits on each square side forcing any tiling to be the

[1] The reader may want to show this extension: Theorem: no Turing machine algorithm exists which will always decide in $< F$ steps whether a given Turing machine with a given input string will halt in $< F$ steps (where the integer $F$ is also part of the input). In other words, algorithms exist which "can't be speeded up."

usual rectangle grid but with "matching conditions" among neighboring tiles. A horizontal cross section of the rectangle tiling will represent a Turing machine's tape and its head. The vertical direction will represent "time." The tile matching conditions may be used to program any transition function getting the tape and head at time $t$ to time $t+1$. The tiling continues forever from some initial state (i.e. horizontal row of tiles) if and only if the Turing machine never halts for that input. Otherwise, the tiling "gets stuck." The starting tile will represent the head, and it joins to tape character tiles of two types – those to the right and left of the head. One needs to make special head and tape tiles for time $t = 0$ which are "bidirectional" (time goes forward both going down from and up from this horizontal row). Another little hack: we need special "head with character" tiles and "head is really 1 to the left with character" tiles (which have to link to a real head one to the left) to allow us to force head motion. After these kludges, tiling is then equivalent to "an input exists which causes this Turing machine to run forever." That may be seen to be undecidable. QED.

**Collatz's** original $3x + 1$ problem: Start with a positive integer $x$. If $x$ is even, divide it by 2. If $x$ is odd, replace $x$ by $3x + 1$. QUESTION: does this process always end up in the cycle $1 \to 4 \to 2 \to 1$? Enormous numbers of computer experiments and approximate theoretical models suggest the answer is "yes." But the problem remains open.

A generalization of this problem is: depending on the remainder $r$ of $x$ mod $M$ (for some fixed $M$, here 2), replace $x$ by $f_r(x)$, where the $M$ functions $f_1$, $f_2$, etc. are some set of linear functions. The INPUT is: $M$ and a description of $f_1$, $f_2$,... $f_M$. The OUTPUT is: a bit indicating whether this process will always reach $x = 1$ starting from any positive integer $x$. This generaliztion is undecidable.

**Proof sketch:** Just using $M = 6 = 2 \cdot 3$ allows you to emulate Conway's FRACTRAN, which can emulate any Turing machine. Reaching $x = 1$ happens only if the Turing machine halts, which is undecidable. QED.

**Diophantine equations** problem: [Y.Matiyasevic & J.Robinson: Acta Arith. 27 (1975) 521-553.]
INPUT: symbolic description of some multivariate polynomials with integer coefficients.
OUTPUT: 1 bit, indicating whether a solution exists: a set of integers at which all the polynomials assume value 0.

There are numerous other undecidable problems. Obviously, if we could solve the halting problem, it would be "trivial" to prove or disprove Fermat's last theorem (also a diophantine equation, although of a different kind) and things of that nature, so it is not surprising that the halting problem isn't easy.

### *4.2 Gödel undecidability*

A statement is said to be "Gödel undecidable" if no proof or disproof exists for that statement.
**Example:** Fermat's last theorem is now known to be true, since a proof was found by Andrew Wiles. Hence, it is decidable. Before Wiles's proof was found, for all we knew at the time, it could have been undecidable and true, or it could have been decidable and true. Or, it could have been false – but then it would necessarily have been decidable, since a proof of falsity would exist – simply write down a counterexample (suitable integers $a, b, c, n$ with $a^n + b^n = c^n$). If a proof existed that FLT were undecidable, then FLT would necessarily be true... hence the undecidability of FLT would have had to be itself undecidable!
**Example:** The statement "there are an infinite number of primes of form $n^2 + 1$" could be all four possible combinations of {true, false} and {decidable, undecidable}, so far as is currently known.

Gödel and Turing undecidability are related. If a statement is Gödel decidable, then an algorithm to find the proof settling the statement is simply to search through all possible candidate proofs (strings of characters obeying certain local conditions described by logical "axioms") until you find it.

Hence we conclude that there exist Turing machine programs $P$ such that the statement "$P$ always halts" has no proof or disproof (since if every $P$ had such a proof, then the halting problem would be Turing decidable). Now with just about any interesting system of logical axioms you can define Turing machines (i.e. the transition function is a finite list of logical implications, the halting question is then equivalent to "do these implications allow one to derive (the halt state)?"). Hence,

**Theorem 3** *(Gödel) With essentially any finite set of axioms, there exist Gödel undecidable statements!*

I think Gödel undecidability is not the same as "logical independence." For example, P.J.Cohen and K.Gödel showed the "continuum hypothesis" was *logically independent* of the usual axioms of set theory and the integers. [Continuum hypothesis: a set of larger cardinality than the integers, but smaller than the reals (in the sense of Cantor) exists.] As Cohen put it "The notion of a set is too vague for the continuum hypothesis to have a positive or negative answer." More precisely, the usual axioms plus CH is just as consistent as the usual axioms alone (just as nonEuclidean geometry is just as consistent as Euclidean geometry since you can construct it *inside* Euclidean geometry); the usual axioms plus NOT(CH) is too.

Hence, we could assume the truth or falsity of the continuum hypothesis CH as an additional axiom (just as we may assume the parallel postulate or its falsity in addition to the other axiomns of geometry), and we'll never encounter a contradiction, and, in fact there will *be* no contradiction if the other axioms were consistent.

On the other hand, suppose "there are an infinite number of primes of form $n^2 + 1$" is true, but Gödel undecidable. Well, if we nevertheless assumed it was false – some last such prime exists – and added this to the usual axioms of number theory, we'd get away with using this false assumption forever in the sense that nobody could ever *prove* a contradiction! But... we still assumed something false and everything we did thereafter was bogus. (Although our work would still be useful in the sense that all the consequences we derived and claimed "true," we genuinely did prove to be "either true, or undecidable, or both.")

So... I guess there really is a difference between these two notions, but in practice it is an undetectable difference!

### *4.3 Brouwer's "intuitionism"*

Horrible scenarios like this[2] make one a lot more sympathetic with L.E.J.Brouwer's "intuitionism." Brouwer wanted proofs by contradiction to be forbidden because they depend on the notion – which he rejected – of the "excluded middle:" that every statement is either true or false. Brouwer then tried to redo all of mathematics with purely constructive proofs, avoiding any proofs by contradiction – or classifying theorems of mathematics as "bad" if he could show that no Brouwer-acceptable proof of it could exist. Contradiction remains a Brouwer-acceptable weapon in all arguments about finite sets; troubles are encountered only when infinities enter the picture. Even then, Brouwer showed, proof by contradiction may still be used in certain scenarios. For example, Brouwer, after re-constructing the real numbers, proved the fine theorem that if $a \le b$ is false, then $a > b$ is true! But... sadly, few mathematicians have had the courage to follow Brouwer, and most of them are now dead[3]. I also lack this courage, at least for the purposes of this course. Thus, I've used proof by contradiction freely in these notes; Brouwer would probably have been disappointed.

### *4.4 Higher levels of undecidability*

It gets weirder. Consider a Turing machine equipped with an "oracle" that can solve halting problems posed to it by the TM. Call this class of more-powerful-than-Turing machines "$TM^2$." $TM^2$ machines can instantly solve Fermat's last theorem. It would be rather nice to have one. Well... the halting problem for $TM^2$ machines is undecidable by $TM^2$ machines. You might call problems of this type "second level undecidable." In fact you can define $TM^n$ to be Turing machines equipped with oracles to solve the halting problem for $TM^{n-1}$ machines, leading to an infinite "undecidability heirarchy" of *really* hard problems.

**Go ten times faster?** Henry Cejtin told me once that Turing machines and computers in general would be far finer, if they only had one more instruction, the "go ten times faster" (GTTF) instruction. (After executing it, all future instructions go 10 times faster!)

In fact, a Turing machine equipped with this instruction would be capable of doing an infinite amount of computation in finite (in fact constant) time, and hence it could solve the halting problem in 1 minute. (You run the GTTF instruction every other step. The time to execute an infinite number of instructions is $2(1 + 10^{-1} + 10^{-2} + ...) = 20/9$.)

Two communicating such Turing machines, a "master" and a "slave," (the master can reset the slave to the start state, and can write on its input tape; the slave machine, immediately after reset, runs at the same speed as the master) could solve the halting problem for $TM^2$ machines in 2 minutes. More generally, $k$ communicating such Turing machines in socially ranked order (each machine is the master of the next one) could solve the halting problem for $TM^k$ machines in $k$ minutes.

So, this instruction is indeed a powerful one! And this idea is maybe not as dumb as it sounds... since I will use it later to make $N$ bodies moving under Newton's laws of motion perform an "infinite amount of computation" in 1 minute.

## 5 Definition of time and space complexity

The "time complexity" of an algorithm is the function $T(N)$ such that the longest that the algorithm can run, on an $N$-bit input, is $T(N)$ steps.

The "space complexity" of an algorithm is the function $S(N)$ such that the most space (bits of memory; on a Turing machine, the furthest to the right one ever moves along the tape) that the algorithm can consume, on an $N$-bit input, is $S(N)$.

These quantities may differ for different models of computation. Similarly the complexity of a problem class is the complexity of the best algorithm solving it.

## 6 P, NP, and NP-completeness

### *6.1 P: Polynomial time*

A problem class is in "$P$" if its time complexity is bounded by a polynomial function of $N$.

**Example.** Integer squaring is obviously in $P$ (you can square an $N$-digit long integer in polynomial($N$) steps).

**Example.** Integer factoring (given an $N$-digit long integer, find a nontrivial factor) is conjectured not to be in $P$. However, *verifying* that the answer is a factor, is in $P$.

**Example.** Primality testing is not obviously in $P$... but in difficult recent work by Adleman and Huang it was shown to be in "$RP$." That is: there exists a Turing machine equipped with a random bit generator which will, with probability $> 2/3$, in a number of steps bounded by a fixed polynomial($N$), produce a proof or disproof that your $N$-digit integer is prime. With probability $\le 1/3$, it will instead announce "sorry, I got unlucky" at which point you could try testing the same number again with new random bits.

The class $P$ is regarded by many as representing a close approximation to the set of "tractable" problems. It is given added importance due to Church's strong thesis, which

[2] And there's more! "Russell's paradox" is: a barber in a village shaves everybody who does not shave himself. So: who shaves the barber? As a set-theoretic paradox this is: a set has "property P" if it does not contain itself as an element. Now consider the set $S$ of sets with property P. Does $S$ have property $P$? Russell, in trying to write down axiomatic foundations of set theory, squirmed out of this one by simply not allowing sets to contain themselves. Later he tried keeping track of the depth of recursive definitions and not allowing them to get deep. Either way, it was ugly.

Also there was Cantor's paradox arising when Cantor tried to define "cardinalities" of infinite sets: the "powerset" of all subsets of a set may be shown to have larger "cardinality" than the set itself. But then, the set of all sets, which clearly contains itself, has larger cardinality than itself.

[3] D.Hilbert once remarked: "Mathematicians will never give up proof by contradiction, though Brouwer and the intuitionists may speak with the tongues of angels."

claims that polynomial time is polynomial time is polynomial time, no matter what reasonable model of computation you use[4]. This also gives doubly added importance to classifying problems as in P versus being NP complete, since such classifications are often possible and place the problem squarely on one or the other side of the presumed tractable-intractable boundary.

### 6.2 *NP: Problems such that checking the answer is in P*

A problem class is in "NP" if:

1. The solution to each problem in the class is either "yes" or "no" (1 bit).
2. If the answer is "yes," then there is a proof the answer is yes, and this proof is at most polynomial($N$) bits long, and it may be checked for validity by an algorithm in $P$.

. In other words, problems in NP have quickly verifiable yes answers. The problems with quickly (P) verifiable "no" answers are "Co-NP."

**Example.** The yes-no variant of integer factoring is definitely in NP. (INPUT: an integer $X$, and another smaller integer $B$, both positive. OUTPUT: Is there a prime factor of $X$ below $B$: yes or no?) However, it quite likely is not in Co-NP, nor in $P$. Note: if the yes-no variant were in $P$, then the usual variant would also be in $P$ since we could use the yes-no variant to do a binary search for the answer.

**Example.** Primality testing is in NP∩Co-NP.

**Proof it is in Co-NP:** If a number is non-prime, there is a short proof of that (just give a factor).

**Proof it is in NP:** If a number $M$ is prime, there is also a short proof of primality... this is trickier. The idea is to factor $M-1$ completely into primes $p_1$, $p_2$, ... $p_k$ (proving the primes in the factorization are valid recursively). Then we may apply the theorem that $M$ is prime if and only if there exists a "generator" (integer $g$ with $1 < g < M-1$) such that $g^{M-1} = 1 \bmod M$, but $g^{(M-1)/p_j} \neq 1 \bmod M$ for all $p_j | M-1$. So the proof of primality is: exhibit the factorization of $M-1$, the recursive proofs of primality, and a generator $g$. If $M$ is $N$ bits long, then the primality proof is checkable in polynomial($N$) steps, and indeed is polynomial($N$) bits long at most. Why quickly checkable? Because the computation of $g^{M-1} \bmod M$ may be done in polynomial time by "binary powering." (Exercise: show how to compute $a^b$ using $O(\log b)$ multiplications.) Why concise? Because: Suppose the primality proof for an $N$-bit number is $B(N)$ bits long. Then

$$B(N) \leq N + \sum_{j=1}^{k} B(\log_2 p_j) \leq 2N + \sum_{j=1}^{k} B(N_j) \qquad (2)$$

where $\sum_{j=1}^{k} N_j = N$ and $0 \leq N_j \leq N-1$. Well, we may conclude from this recurrence that $B(N) \leq N^2$, which is a polynomial($N$). QED.

[4] Sort of a collaboration between Alonzo Church and Gertrude Stein.

### 6.3 *NP-completeness*

A problem class X is "NP-complete" it is in NP and if any other problem in NP may be rephrased as a problem in class $X$, in such a way that this rephrasing only requires at most polynomial expansion of the number of bits in the input ($N$ bits goes to polynomial($N$) bits) and only requires at most polynomial time for a Turing machine algorithm to perform the rephrasing.

**Theorem 4** *NP-complete problems exist!*

**Proof.** (S.Cook.) Consider the "Boolean circuit satisfiability" (or "SAT") problem:
INPUT: a combinatorial description of a Boolean feedforward-only circuit (made of 2-input NAND gates and wires) with $N$ input bits and one output bit.
OUTPUT: Does there exist a way to choose the values of the $N$ input bits, so that the circuit's output would be '1'?

This problem is clearly in NP (verifying that the magic input settings actually work, once you know them, is easy). But for *any* problem whose answer is quickly verifiable (i.e. which is in NP), you could build a polynomial size Boolean circuit to perform that verification. For example to verify an integer factorization, you'd build a multiplier circuit followed by a $(=, \neq)$ comparator. If this claim is not obvious to you, consider a circuit whose $t$th "layer" performs a Turing machine step at time $t$, and the inputs and outputs of that layer are bits representing the head and tape states at that moment in time. Thus a circuit with $T(N)$ layers, each $O(T(N)+N+1)$ bits wide, will suffice to simulate any Turing machine on $N$-long input running for $T(N)$ steps. Hence any problem in NP is easily rephrasable as a Boolean circuit satisfiability problem. QED.

NP-complete problems are, up to polynomial slowdown, the hardest problems in NP. "NP-hard" problems are problems which are NP-complete except that they are not necessarily in NP, because, e.g., they are not yes-no problems. All NP-complete problems are automatically NP-hard, but NP-hard problems could be harder. That is, any problem in NP may be rephrased as your NP-hard problem, and the rephrasing is in $P$; so if you had a friend who could solve NP-hard problems on demand, you could with only polynomial work, solve *any* problem in NP in polynomial time.

*The biggest open problem in computer science is: is $P \neq NP$ (as is widely believed)?* This would be implied by $Co-NP \neq NP$; it would also be implied by any proof that any NP-complete problem is superpolynomially hard for worst case input.

There are numerous NP-complete problems; the book by Garey and Johnson contains a list of over 300 of them.

**Example.** "3-SAT" is NP-complete. (INPUT: a set $S$ of 3-variable Boolean OR clauses such as $x_3 + \overline{x_7} + x_9$, or $x_5 + x_6 + x_7$, or $x_1 + \overline{x_7} + \overline{x_8}$ where + means OR (and overlines mean Boolean negation) and the subscripts are selected from a finite set $\{1, ..., N\}$. OUTPUT: Does there exist a set of Boolean values $x_1$, $x_2$, ..., $x_N$ such that all of the clauses in $S$ are TRUE?)

**Proof:** Plainly 3-SAT is in NP (it is easy to verify a truth assignment that works). Associate a Boolean variable $x_k$ with the $k$th wire in any Boolean circuit with $N$ wires. The relations enforced by each NAND gate in the circuit is expressible as the AND of a finite set of ORs of 3 Boolean variables each. (Just write down every possible such OR and discard the ones which can be untrue for some Boolean values of the 3 wires connected to that NAND gate.) So, any feedforward Boolean circuit satisfiability problem may be quickly rephrased as a 3-SAT problem... but *any* problem in NP may be rephrased as the former, and hence as the latter... so 3-SAT also is NP-complete, QED.

**Example.** INDEPENDENT SET is NP-complete. (INPUT: a description of a graph $G$ [set of edges and vertices] and an integer $B$. OUTPUT: One bit indicating whether an "independent set" (set of vertices, no two adjacent) exists in $G$, with cardinality $\geq B$.)

**Proof:** Clearly this problem is in NP: given an independent set, the cardinality bound and the non-adjacency relationships are trivially verifiable in polynomial time. We'll show that any 3-SAT problem instance with $N$ input bits (and hence any problem in NP) and $C$ clauses may quickly be rephrased as an INDEPENDENT SET problem instance on a graph with $3C$ vertices.

The graph has a vertex for every ordered pair $(v, c)$ where $v$ is a variable (from the $2N$ element set $\{x_1, x_2, ..., x_N, \overline{x_1}, \overline{x_2}, ...\overline{x_N}\}$ in clause $c$). Thus there are $3C$ vertices. Two vertices $(v_1, c_1)$ and $(v_2, c_2)$ are joined by an edge in the graph if $c_1 = c_2$ or $v_1 = \overline{v_2}$.

The joinings of $x_j$ to $\overline{x_j}$ assures that at most one of these is in an independent set, i.e. in the logical version, that at most one of a variable and its negation is TRUE. The joinings of the 3 vertices corresponding to the same clause in a triangle causes at most one vertex per clause to be in the independent set, and this is achievable if and only if some assignment of Boolean values causes every 3-SAT clause to be TRUE. QED.

**Remark.** The complement of an independent set is a "cover," that is, a set of vertices such that every edge in the graph contains at least one vertex from our coverset. Also, an independent set in the *complement* graph is a "clique," i.e. a set of vertices every two of which are adjacent in the complement graph. Thus these problems (existence of a vertex cover of cardinality $\leq B$, or existence of a clique of cardinality $\geq B$, in a graph) are also NP-complete and any one of them is easily transformed into any other.

**Example.** It is NP-complete to find out if the "ground state" of a "spin glass" is below some energy threshhold $E$ or not. (INPUT: a set of integer "interaction energies" $U_{jk}$ for $1 \leq j, k \leq N$, and an integer $E$. OUTPUT: Does there exist a set of $N$ "spin values" $x_k \in \{+1, -1\}$ so that $\sum_{j=1}^{N} \sum_{k=1}^{N} U_{jk} x_j x_k < E$?)

**Proof sketch.** Plainly this is in NP. One may see that the INDEPENDENT SET problem on a graph with $N-1$ vertices is a special case of SPIN-GLASS, so it is in fact NP-complete. QED.

### *6.4 The polynomial heirarchy*

Imagine a Turing machine equipped with an oracle which will solve any 3-SAT problem posed to it in 1 step. This machine's polynomial time capabilities (assuming $P \neq NP$ and Church's strong thesis) exceed any reasonable computer's. Call the class of problems this machine can solve in polynomial time $P^2$. The class NP$^2$ is then the class of problems with yes-no answers which whose yes-answers may be verified in $P^2$. There are NP$^2$ complete problems such as "SAT$^2$"
INPUT: a description of a $P^2$ algorithm (e.g. as a Boolean feedforward circuit with oracle boxes, $N$ Boolean inputs and one Boolean output).
OUTPUT: a bit indicating whether an input to the circuit exists which will cause the output to be '1.'

Similarly you can consider $P^k$ and $NP^k$ for $k = 2, 3, 4, ....$ ($P^k$ allows oracle calls to solve $NP^{k-1}$-complete problems.) This is the "polynomial heirarchy" (in a nonstandard notation).

### *6.5 #P, the class that counts*

"#P" (pronounced "number-P") is the class of problems of the form
INPUT: a description of a $P$ algorithm (e.g. as a Boolean feedforward circuit with $N$ Boolean inputs and one Boolean output).
OUTPUT: an $N$-bit integer *counting* how many inputs to the circuit cause the output to be '1.'

A problem complete for #P is "counting SAT" (which is actually the problem we just described). By "complete" we mean: any counting problem in #P may be rephrased as a counting SAT problem with the same count as output and such that the rephrasing process takes polynomial time.

A different problem complete for #P is "INTEGER PERMANENT:" the input is an integer $N \times N$ matrix and the output is the integer permanent of that matrix. (This was shown by L.G.Valiant: Theor. Comput. Sci. 8 (1979) 410-421.)

#P clearly includes NP. The relation between #P and the polynomial heirarchy is unclear.

Almost certainly #P≠NP... but this has never been proven.

### *6.6 PSPACE*

PSPACE is the class of problems soluble by a Turing machine using polynomial($N$) memory space at most (that is, its head never moves further than polynomial($N$) to the right of the start position) on $N$-bit inputs.

It is obvious that #P⊆PSPACE since you could just loop over all $2^N$ possible configurations of $N$ input bits to a Boolean circuit and count how many of them satisfy it, while consuming polynomial($N$) memory space.

It is generally conjectured that #P≠PSPACE.

An example of a PSPACE-complete problem (any problem in PSPACE may be rephrased as an instance of this problem, and the rephrasing may be found by a PSPACE algorithm) is the **"bounded-space halting problem:"**

INPUT: A description of a Turing machine with a bounded (not semi-infinite as usual) tape containing its input.
OUTPUT: Would that Turing machine halt on that input?

A different PSPACE-complete problem is solving the following game.
INPUT: A directed graph $G$ with a distinguished "start vertex" $v$.
RULES OF GAME: Two players alternate moves. Each move, you get to move the current location along any outgoing directed edge to a new vertex. After each such move, you "burn your bridges behind you," i.e. erase the directed edge you just walked across. If you can't move, you lose.
OUTPUT: A bit indicating whether the first player can force a win.

### 6.7 EXPTIME

It's generally believed that PSPACE-hard and NP-hard problems require exponential(polynomial($N$)) solution time for worst case input, but this has never been proven. It gets to be rather annoying that nobody can prove these "obvious" hardness conjectures. However, we are not completely helpless. We *can* prove[5] the exponential hardness of "EXPTIME-complete" problems, and it is this that makes EXPTIME particularly interesting.

EXPTIME is the class of problems soluble by a Turing machine in $\leq$exp(polynomial($N$)) steps at most for $N$-bit inputs.

It is obvious that PSPACE$\subseteq$EXPTIME since in EXPTIME you could explore every possible state of the PSPACE machine!

It is generally conjectured that PSPACE$\neq$EXPTIME.

An example of an EXPTIME-complete problem (any problem in EXPTIME may be rephrased as an instance of this problem, and the rephrasing may be found by an EXPTIME algorithm) is the **"bounded Turing machine game:"** [A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer: Alternation, J. ACM 28,1 (1981) 114-133.]
INPUT: A description of Turing machine with a bounded tape, and the input written on that tape. The Turing machine has two 1-bit outputs and a bit input. Consider the following game. Every time the Turing machine activates output 1, player $j$ (where $j$ is the value of output 2) chooses a value of the input bit. If and when the Turing machine halts, its output bit 2 tells us who the winning player is.
OUTPUT: Can the first player force a win?

Here is another EXPTIME-complete problem that perhaps you'll like more[6] – **checkers!**
INPUT: An $N \times N$ board position in $N \times N$ "checkers."
OUTPUT: Can white force a win?

To prove EXPTIME-complete problems really are exponentially hard, we need to exhibit *a* problem in EXPTIME which is exponentially hard. The following problem, **"the additive theory of the reals (or rationals),"** is known to require $\exp(cN)$ steps to solve, for an infinite number of inputs, for some constant $c > 0$ [9]. [See also: Anna R. Bruss & Albert R. Meyer: On time-space classes and their relation to the theory of real addition, Theor. Computer Sci. 11,1 (1980) 59-69; Jeanne Ferrante & Charles Rackoff: A decision procedure for the first order theory of real addition with order. SIAM J. Computing 4,1 (1975) 69-76; Leonard Berman: The Complexity of Logical Theories, Theor. Computer Sci. 11 (1980) 71-77; M.J.Fischer & M.O.Rabin: Super-Exponential Complexity of Presburger Arithmetic, Proc. SIAM-AMS Sympos. in Applied Math. 7 (1974) 27-41; Derek C. Oppen: A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic, J. Computer & System Sci. 16,3 (1978) 323-332; A. R. Meyer & L. J. Stockmeyer: The equivalence problem for regular expressions with squaring requires exponential space, 13th IEEE Symposium on Switching & Automata Theory (1972) 125-129.]
INPUT: A finite "sentence" involving the symbols $+$, $=$, $<$, 0, 1, parentheses, variables $x_1$, $x_2$,..., and logical connectives $\exists$, $\forall$, AND, OR, and NOT.
OUTPUT: Is this sentence (assuming it passes simple grammatical tests so that it has a meaning) a true fact about real (or rational) $x_j$ – yes or no?
**Example.** The sentence $\exists y \forall x (x \leq y)$ makes sense, but is false (there is no greatest real or rational). The sentence

$$\forall x \, \forall y \, [x = y \text{ OR } \exists z (x < z \text{ AND } z < y) \\ \text{ OR } \exists z (y < z \text{ AND } z < x)] \tag{3}$$

is true (between any two unequal reals, there is a third real).
**Remark.** The similarly defined additive theory of the *integers* is known to be decidable and known to require *doubly* exponential time $\exp 2^{cN}$. If we put multiplication in also, the theory of the reals remains decidable, but the theory of the integers is undecidable.

## 7 A FEW GOOD ALGORITHMS

Computer scientists have developed many clever algorithms, which run a lot more quickly than one would naively have supposed possible. Lots of them are listed in algorithms books such as [4] [10].

In some cases, they have also managed to prove *lower bounds*, showing, for example, that *any* algorithm, running on some model $C$ of computation, which solves all $N$-bit long problems from class $X$ and always halts in time $T(N)$, must obey $T(N) > B(N)$. However, there has been much less success in proving lower bounds than there has been in proving upper bounds – as is evidenced by the (pathetic) fact that nobody can prove the "obviously true" conjecture that $P \neq NP$.

### 7.1 Sorting in $O(N \log N)$ steps

We are on a real (or integer) RAM. We are given $N$ items as input. The task is to sort them into order.

[5] Well, this is what it says in numerous CS literature sources. But: I admit that I've nowhere seen an actual explicit exponential-type function and an explicit problem on an explicit model of computation with "this problem requires more solution time than this function for an infinite number of inputs." Such an explicit bound of the form $\exp(cN)$ for an explicit $c > 0$, ought to be deducible from [9].

[6] J.M. Robson: NxN checkers is EXPTIME complete, SIAM J. Comput 13,2 (1984) 252-267.

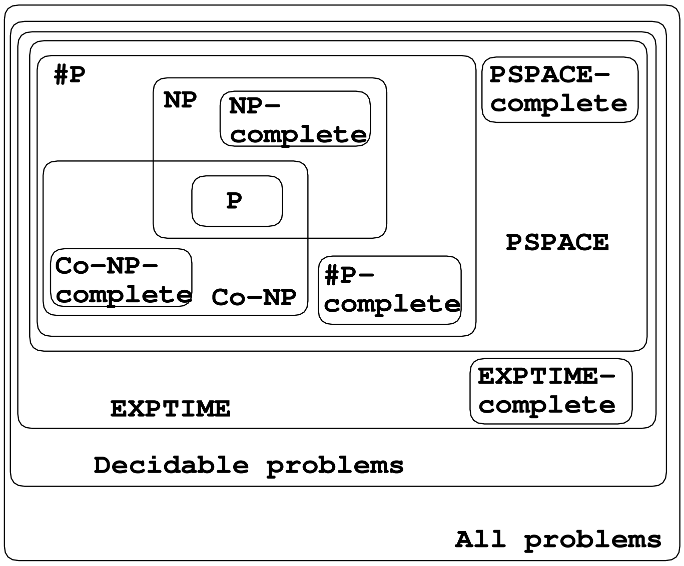


Figure 2: Some important complexity classes and their (believed) inclusion relations.

The naive method of finding the minimum, removing it, finding the minimum of what is left, removing it, and so on, would seem to require order $N^2$ steps. Can we do better?

**Theorem 5** *(Upper bound for sorting.) The sorting problem above may be solved in $O(N \log N)$ steps (indeed, $\le N \log_2 N$ item $\{<, \ge\}$ comparison operations) on a RAM permitting item copying, array access, and item comparisons, each in 1 step.*

**Proof.** Here is one method called "merge sort." One can merge two sorted arrays of $A$ and $B$ items, creating a new sorted array of $A+B$ items, in $O(A+B)$ steps, indeed in $\le A+B-1$ pairwise comparison operations. (You walk through the two sorted arrays comparing the two current elements, copying the smaller one into the new merged array, and advancing in the appropriate subarray to "walk past" the used-up element.)

So we start with $N$ arrays each with 1 element, merge them in pairs to get $N/2$ arrays each with 2 elements (sorted), merge them in pairs to get $N/4$ arrays each with 4 elements (each sorted) and so on. QED.

**Theorem 6** *(Lower bound for sorting.) Any algorithm for the sorting problem above must make at least $\log_2(N!)$, which by Stirling's formula is asymptotic to $N \log_2 N$, element-pair comparisons. (Hence mergesort's comparison count is asymptotically optimal in the worst case.)*

**Proof.** The same output could have appeared in the input in any of $N!$ permuted forms. After $k$ elementwise comparisons have been done, the program has made $2^k$ or fewer choices. So if fewer than $\log_2(N!)$ comparisons were performed during the sorting, then the algorithm must have done the same thing (i.e. produced the same reordering) for at least two of the $N!$ possible inputs, i.e. must have done the wrong thing on at least one of them. QED.

### 7.2 *Matrix multiplication and inversion (and solving linear systems)*

The naive method for multiplying two $N \times N$ matrices requires order $N^3$ scalar arithmetic operations. Gaussian elimination and its ilk also require order order $N^3$ operations to solve linear $N \times N$ systems or invert an $N \times N$ matrix.

Can we do better? The key trick, discovered by V.Strassen and refined by S.Winograd and me, is that if

$$\begin{pmatrix} W & X \\ Y & Z \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} E & F \\ G & H \end{pmatrix} \tag{4}$$

then we may compute $W$, $X$, $Y$, and $Z$ by the following code fragment

```
s1 = C+D; s2 = F-E; Y = s1-A;
X = H-s2; s3 = Y*X; W = A*E; s4 = B*G;
s3 += W; W += s4; X -= G;
s4 = D*X; Z = B-Y; X = Z*H; Z = s1*s2;
s2 = A-C; s1 = H-F; Y = s2*s1; X += s3;
Y += s3; X += Z; Z += Y; Y -= s4;
```

note "C-language" notation: $f+ = g$ is a shorthand for "add $g$ to $f$." Note, this algorithm works even if the "*" operation is *noncommutative.* The good news is, this multiplies two $2 \times 2$ matrices using only 7 scalar multiplies (as opposed to the naive 8). The bad news is, it uses 15 $\pm$ operations (versus than the usual 4)[7].

**Theorem 7** *Two $N \times N$ matrices may be multiplied in $\le 4.8 N^{\log_2 7} < 4.8 N^{2.8074}$ arithmetic operations, using storage for $4N^2/3 + O(\log N)$ numbers beyond the storage needed to represent the input and output numbers. The number of arithmetic operations always is smaller than the classical algorithm if $N \ge 14$.*

**Proof sketch.** Apply our code fragment in a divide and conquer fashion. If $N$ is even, use the identity to reduce to 7 problems of size $N/2$, plus $O(N^2)$ work. If $N$ is odd, deal with the last row and column of the matrices by brute force after a reduction to 7 problems of size $(N-1)/2$; note extra work is still $O(N^2)$. But if $N < 14$, use the naive matrix multiplication algorithm and don't reduce to subproblems. If $T(N)$ is the number of arithmetic operations needed to multiply two $N \times N$ matrices, we get recurrences such as

$$T(N) \le 7T(\frac{N}{2}) + O(N^2) \tag{5}$$

with solution $T(N) = O(N^{\log_2 7})$. The recurrence for the excess storage is

$$S(N) = S(\frac{N}{2}) + 4(\frac{N}{2})^2 + O(1) \tag{6}$$

[7] Strassen found how to achieve 7*, which was shown to be best possible by J.Hopcroft & R.Kerr: SIAM J.Appl.Math. 20,1 (1971) 30-36. Winograd found how also to achieve 15$\pm$, which was shown best possible by R.Probert: SIAM J. Comput. 5,2 (1976) 187-203. See also S.Winograd: Lin. Algeb. Applic. 4 (1971) 381-388. I showed that the assignment ordering given in the code fragment above leads to the minimum possible amount (4 variables `s1,s2,s3,s4`) of temporary storage. It remains possible there's some inequivalent 7*,15$\pm$ method requiring less temporary storage, but I conjecture not.

with solution $S(N) = 4N^2/3 + O(\log N)$. QED.

[J.R.Bunch & J.E.Hopcroft: Math. of Comput. 28 (1974) 231-236] showed that

**Theorem 8** *LUP matrix factorization (L lower triangular, U upper triangular, P a permutation matrix) and matrix inverse are equivalent in arithmetic operation-count complexity to matrix multiplication (up to constant factors).*

**Proof.** The block matrix identity

$$\begin{pmatrix} I & A & 0 \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}^{-1} = \begin{pmatrix} I & -A & AB \\ 0 & I & -B \\ 0 & 0 & I \end{pmatrix} \tag{7}$$

shows how to use a matrix invertor to do matrix multiplication, and proves the later task is at worst a constant factor more expensive than the former. On the other hand,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ A_{21}A_{11}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{11} & 0 \\ 0 & \Delta \end{pmatrix} \begin{pmatrix} I & A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix} \tag{8}$$

where $\Delta = A_{22} - A_{21}A_{11}^{-1}A_{12}$. This latter formula allows LU factorization of $A$, if the 3 recursively done half size matrix inversions are performable, and $A^{-1}$ may be deduced via

$$A^{-1} = \begin{pmatrix} I & -A_{11}^{-1}A_{12} \\ 0 & I \end{pmatrix} \begin{pmatrix} A_{11}^{-1} & 0 \\ 0 & \Delta^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -A_{21}A_{11}^{-1} & I \end{pmatrix} \tag{9}$$

under similar assumptions of invertibility. (Bunch & Hopcroft show how to perform "pivots" [permutations] to avoid problems with noninvertible smaller matrices. This would allow us to drop these assumptions.) One may see from this that we can perform matrix factorization or inversion at worst a constant factor more slowly than matrix multiplication. The equivalence, to within a constant factor, of matrix multiplication and matrix inversion, has now been established. QED.

**Remarks.** Ibarra & Moran in J.Algorithms 3,1 (1982) 45-56 considered fast "least squares" solutions to systems of linear equations, and factorizations of rectangular matrices. D.Coppersmith & S.Winograd: J. Symbolic Computation 9,3 (1990) 251-280, found a (highly impractical) matrix multiplication algorithm requiring only $O(N^{2.376})$ arithmetics, which is the best known exponent at the moment.

K.Mehlhorn in [10] gave analysis showing that the *bit complexity* (as opposed to arithmetic operation count) of Strassen is also an improvement over classical, thus partly giving the lie to claims of numerical instability.

### *7.3 Fast fourier transform*

The "discrete fourier transform" (DFT) is the computation of

$$b_k = \sum_{m=0}^{N-1} a_m \exp \frac{\pm 2\pi i m k}{N} \tag{10}$$

for $k = 0, ..., N-1$ from $a_0, ..., a_{N-1}$. ($\vec{a}$ and $\vec{b}$ are complex.) Here use $+$ for the "forward" DFT and $-$ for the backward DFT. If you do a forward DFT followed by a backward DFT (and then scale by $1/N$) you get the original $N$-vector back.

Computing a DFT may be thought of as evaluating the polynomial of degree $N-1$ whose coefficients are the $a_m$, at $N$ points on the unit circle in the complex plane forming the vertices of a regular $N$-gon. Naively, doing this would seem to require at least order $N^2$ arithmetic operations. Can we do better?

**Theorem 9 (Fast Fourier Transform, or "FFT")** *The DFT (EQ 10) may be computed using* $2N\log_2 N + 2N + O(\log N)$ *real multiplications, and* $3N \log_2 N + 2N + O(\log N)$ *real additions. (Also at most* $2N \log_2 N$ *total subscripting operations are needed.)*

**Proof sketch.** This was actually realized by Gauss and a number of other early discoverers. But, the algorithm only "stayed discovered" once it was given by J.Cooley & J.Tukey: An algorithm for the Machine Calculation of Complex Fourier Series, Math. of Comput. 19 (1965) 297-301.

The basic idea is that to calculate the DFT

$$A_k = \sum_{j=0}^{N-1} a_j W^{kj}, \quad k = 0..N-1, \tag{11}$$

where $N$ is a power of 2, $N > 1$, and $W$ is a principal $N$th root of unity, e.g. $W = \exp(2i\pi/N)$ if we are in the complex numbers, we may write

$$A_k = g_k + h_k W^k \tag{12}$$

where

$$g_k = \sum_{0 \le 2j < N} a_{2j} (W^2)^{kj} \tag{13}$$

$$h_k = \sum_{0 \le 2j < N} a_{2j+1} (W^2)^{kj} \tag{14}$$

are FFTs of half the size (on the even and odd indexed $a$'s, respectively). Note that $g_k$ and $h_k$ are periodic in $k$ with period $N/2$, since $W^{N/2} = -1$.

So if $T(N)$ is the time (or arithmetic op count) to calculate an FFT of size $N$, then $T(N) \le 2T(N/2) + O(N)$ and so $T(N) = O(N \log N)$. (We assume the single number $W$ is already available.) The inverse transform is calculable by the same method using $W^{-1}$ instead of $W$.

Incidentally, the following identity is useful for computing consecutive powers of $W$:

$$e^{ic(k+1)} = e^{ic(k-1)} + 2i \sin(c) e^{ick}; \tag{15}$$

it requires only 2$*$ and 2$+$ operations per complex exponential computed, and it is also numerically stable since it involves the *small* constant multiplier $2i\sin(c)$, which tends to cause numerical errors to die off exponentially. This is as opposed to the slower, and also unstable, naive method

$$e^{ic(k+1)} = e^{ic} e^{ick} \tag{16}$$

in which numerical errors tend to grow exponentially.

My implementation of the complex FFT in the C language typically yields 500 times smaller numerical error than the naive $O(N^2)$ time algorithm when doing a 128 point forward followed by a 128 point backward transform, while being over 10 times faster.

Another kind of FFT: Instead of the complex numbers, you can operate over any *finite field* containing an $N$th root of unity $W$ where $N$ is a power of 2. Thus we may work in the land of integers modulo primes of the form $2^a b+1$, where $N = 2^a$. This allows exact arithmetic (no roundoff error)... although the precision requirements are rather large.

**Fast integer multiplication:** The DFT *convolution theorem* allows one to multiply two $N$-digit integers in approximately $O(N \log N)$ steps, which is a big improvement over $O(N^2)$ from the "shift and add" multiplication algorithm one is taught in school. The idea here is that each digit in the product is expressible as a convolution of digits in the multiplicands, plus "carries" from nearby digits. The convolution may be done by means of two forward FFTs on vectors padded with zeroes, multiply the FFT output vectors elementwise, then do a backward FFT. Another way to think of this: $N$-digit integers are really just polynomials in the radix. To multiply two polynomials of degree $N$, multiply them at $N$ points (the DFT regular $N$-gon points!) and interpolate the resulting data by performing a backward DFT.

Carries only need to be propagated a distance $O(\log N)$ and a numerical error analysis shows that using $O(\log N)$ decimals in the real numbers inside the FFT will suffice to get exact results. Actually the runtime ultimately grows slightly more quickly than $\log N$ since eventually the numbers used in the FFT (and the array subscripts in the FFT) themselves will have to be multiprecision... but even with integers so huge that they totally fill the memory of modern computers, double precision suffices, so $N \log N$ is in practice correct.

Fast integer multiplication (and division, square root, etc.) are just a few of many amazing algorithmic applications of FFTs. Some more are discussed in Borodin & Moenck: Fast modular transforms, J. Computer Syst. Sci. 8 (1974) 366-386; Aho,Stieglitz,Ullman: Evaluating Polynomials at fixed sets of points, SIAM J. Comput. 4,4 (Dec 1975) 533-539; Rabiner,Schafer,Rader: The Chirp-Z transform and its Applications, Bell Syst. Tech. J. 48,3 (1969) 1249-1292.

### *7.4 Algorithmic design paradigms*

Both the FFT and merge sort were examples of the "divide and conquer" algorithmic design paradigm: you divide your problem into subproblems, solve the subproblems recursively, and then combine the subproblem solutions to get a full solution.

Three other algorithmic design paradigms which can often be effective are: problem size reduction, "balancing," and "dynamic programming."

In balancing, which might better be called "minimizing runtime over some variable parameter subject to constraints," you design an algorithm that combines two different algorithmic approaches, and with a parameter governing how much of each approach is "mixed in." Then you "balance" it by choosing the best value of the parameter. For example, one approach might be fast but consumes order $N^3$ memory space to store some big tables. Meanwhile perhaps the other approach only requires order $N$ memory space, but runs slowly since it can't do table lookups. If you can somehow divide your problem into $N^{2/3}$ subproblems each of size $N^{1/3}$, solve each subproblem with the fast method, and combine the subproblem answers, you might get a solution maybe $N^{1/3}$ times faster than if you'd just used the slow method, and *still* you only consume the same amount $O(N)$ (up to a constant factor) of memory space.

### *7.5 "Dynamic programming"*

"Dynamic programming" is a vaguely defined concept. The basic theme is to build tables of data to permit you to get results faster. The tables give solutions to subproblems; small subproblems are solved first and large ones are solved with the aid of the tables of smaller ones.

**Example: path counting.** Suppose you have a graph with two distinguished "source" and "destination" vertices. You want to find out how many paths of length $\le L$ edges there are, to get from the source to destination.

The naive approach would be to explore all possible paths of length $\le L$, adding 1 to a counter if the endpoint of the path happened to be the destination. There might be an exponentially large number of such paths (easy exercise: construct a graph with $\le 10L$ vertices but $\ge 10^L$ such paths), so this algorithm could be extremely slow.

However, there is a much better way. We maintain a table associating with each vertex, the number of $k$-step paths from the source to that vertex. Initially $k = 0$ and all table entries are 0, except that the source itself has table entry 1. To get table $k+1$ from table $k$, simply add up all the counts from each vertex's possible predecessors. For a $V$-vertex, $E$-edge graph, this takes $O(V + E)$ arithmetic operations (additions) to build table $k+1$ from table $k$. Hence we get done in $O(V + E)L$ steps total, and get the answers for *all* possible destination vertices simultaneously as a bonus! At the minor cost of maintaining tables of $O(V)$ entries, we've reduced exponential time to polynomial time.

Actually: we can go further in this direction. The enthusiastic reader may want to figure out how to solve this problem in only $O(V + E) \log L$ steps, at the cost of maintaining larger tables with $O(V^2)$ entries.

**Example: Edit distance.** Given two strings of lengths $A$ and $B$, how "closely related" are they? Problems of this nature come up in DNA studies. One distance metric is "edit distance" – the number of character inserts, deletes, and overwrites needed to transform string 1 into string 2. (Even better: each such operation can be assigned a positive integer "cost," possibly depending on which characters were involved, and we want the sequence of editing operations of least summed cost.)

The naive method considers all possible edit operation sequences and takes a huge amount of time. Forget it.

A better method takes advantage of the *prefix optimality lemma* that any prefix of the first string must have been transformed *optimally* into whatever prefix of the second string it got turned into. (If not, then we could reduce overall cost.)

So we build a table with $AB$ entries. Table entry $T_{ab}$ gives the least cost to transform the first $a$ characters in the first string into the first $b$ characters of the second string. We start with $T_{11}$. Assuming we already know a square block of $T_{ab}$ with $1 \leq a, b \leq k$, we may proceed to enlarge the square block by 1 to get $k \to k+1$. The min cost method corresponding to $T_{ab}$ must, by the prefix optimality lemma, consist of $T_{a,b-1}$ and an insertion (or with a right-end deletion omitted), or it is $T_{a-1,b}$ and do nothing, or it is $T_{a-1,b-1}$ and a change (or non-change). You can consider all these possibilities and choose the min-cost one (minimizing over all possible insertions, changes, etc., which is easy since there is only *one* of them being considered, so just consider all possibilities for it by brute force).

With a fixed alphabet size, the cost is $O(1)$ per table entry, or in total (to build the full table and compute the total cost $T_{AB}$) $O(AB)$ arithmetic operations.

### 7.6 *Information theory, channel capacity, Shannon's theorem*

The "entropy" of a probability distribution with probability $p_j$ of event $j$ is

$$\sum_j p_j \log_2 \frac{1}{p_j} \text{ bits.} \tag{17}$$

One interpretation of this is as follows. Suppose I transmit an $N$ character message to you, $N$ huge, but subject to the constraint that $p_j N$ of the characters in my message must be letter#j. $\text{Log}_2$ of the total number of messages I can transmit is

$$\log_2 \frac{N!}{\prod_j (p_j N)!} \sim N \sum_j p_j \log_2 \frac{1}{p_j} \tag{18}$$

by Stirling's formula $\log N! = N \log N - N + O(\log N)$.

"Information theory" is concerned with the transmission of information through a "noisy channel" between a "sender" and a "receiver."

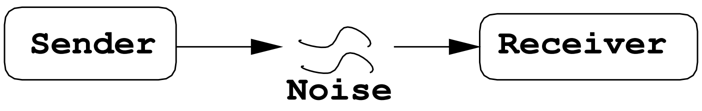


Figure 3: Shannon information transmission picture.

The information consists of a $N$-character message where each character is selected from an $L$-letter alphabet. Suppose channel noise causes a character $i$ sent by the sender to change into another character $j$ received by the reciever, with some "transition probability" $p(j|i)$. Thus the noisy channel can destroy information. The sender and receiver can try to compensate for this by using error-correcting codes.

The "capacity" $C$ of the channel is

$$C = \min_{\substack{0 \leq Q_1, \ldots, Q_L \leq 1, \\ \sum_{i=1}^{L} Q_i = 1}} \; \max_{k \in \{1,2,\ldots,L\}} \tag{19}$$

$$\left( \sum_{j=1}^{L} P(j|k) \log_2 \frac{P(j|k)}{\sum_{i=1}^{L} Q_i P(j|i)} \right).$$

Note that the determination of $C$ is a minimization of a concave-up function within a convex polytope in an $(L-1)$-dimensional space. Such "convex programming" problems can be solved to high precision in time polynomial in the size of the problem[8].

**Example** The "binary symmetric channel" has a 2-character alphabet {0,1} and the noise toggles the bit with a constant probability $E$. So $P(0|0) = P(1|1) = 1 - E$ and $P(0|1) = P(1|0) = E$. The "$\max_k$" has the effect, due to symmetry, of forcing $Q_0 = Q_1 = 1/2$ in the $\min_Q$. The channel capacity is

$$C = 1 + E \log_2(E) + (1 - E) \log_2(1 - E) \tag{20}$$

which goes to 1 when $E \to 0$ (1 bit per bit!) and goes to 0 when $E \to 1/2$ (total noise⇒zero capacity). (Note if $E > 1/2$ we get the same result as if you used $1 - E$.)

**Shannon's theorem** states that *an error-correcting coding scheme (mappings between numbers and $N$-letter messages) exists, so that the sender can transmit a message representing a number in the set $\{1, \ldots, 2^{(C-\epsilon)N}\}$, for any fixed $\epsilon > 0$, and have confidence approaching 1 (as $N \to \infty$) that the receiver will be able to determine the number. However, if the sender tries to transmit a number picked randomly from a uniform distribution in the set $\{1, \ldots, 2^{(C+\epsilon)N}\}$, for any fixed $\epsilon > 0$, then, no matter what scheme is used, with probability approaching 1 (as $N \to \infty$) the receiver will be unable to determine the number. (Both approaches to probability 1 are exponentially rapid.)*

Similarly to the second law of thermodynamics, Shannon's theorem is only a statistical statement; it is conceivable the channel could suddenly decide to create noise far above or far below its usual level, just when you try to transmit an important message. Again, however, one is usually concerned with values of $N$ so large that this is virtually impossible (exponentially unlikely).

**"Shannon's bounds"** can enable one to estimate the capacity of a channel without actually computing it. They state that: Choose any $L$ values $Q_1, Q_2, \ldots, Q_L$ with $0 \leq Q_i \leq 1$, $\sum_{i=1}^{L} Q_i = 1$. Then

$$C \leq \max_{k \in \{1,2,\ldots,L\}} \left( \sum_{j=1}^{L} P(j|k) \log_2 \frac{P(j|k)}{\sum_{i=1}^{L} Q_i P(j|i)} \right), \tag{21}$$

and

$$C \geq \sum_{k=1}^{L} Q_k \left( \sum_{j=1}^{L} P(j|k) \log_2 \frac{P(j|k)}{\sum_{i=1}^{L} Q_i P(j|i)} \right). \tag{22}$$

Both of these bounds are tight if the $Q_i$ are chosen to lie at the optimum of the convex programming problem

[8] P.M. Vaidya: A new algorithm for minimizing convex functions over convex sets, In 30th IEEE Symp. on Foundations of Computer Science (1989) 338-343.

(19). If the transition probabilities have the property that $\sum_{i=1}^{L} P(j|i) = 1$ for each $j = 1, 2, \ldots L$ (e.g. this must happen if the transition matrix is symmetric) then Shannon's lower bound on capacity (with $Q_i = 1/L$) may be written

$$C \geq \log_2 L - \frac{1}{L} \sum_{k=1}^{L} S_k, \quad S_k = -\sum_{j=1}^{L} P(j|k) \log_2 P(j|k). \tag{23}$$

This inequality is in fact an equality if it is also the case that all the $S_k$ are equal. In words: "the capacity of a symmetric channel is the entropy of the source ($\log_2 L$) minus the entropy of the channel."

**Ultra-vague proof sketch:** [5] The existence of good encoding schemes is proven by selecting one at random in a certain manner (note: this existence proof is not very constructive!) and showing that confusion probabilities will then almost always be small. The nonexistence of too-good encoding schemes is proven by a volume argument. You simply cannot pack a lot of points into the space of possible messages, without the "error balls" being highly likely to move you closer to a wrong message than a right one.

## References


[1] S.Arora: Hardness of Approximations, in Approximation Algorithms for NP-hard Problems, PWS Publishing 1996. [Surveys recent extensions to NP theory allowing one to show the NP-hardness of finding *approximate* solutions to many problems.]

[2] C.Bennett: Logical depth and physical complexity, pp. 207-235 in The universal Turing machine, a half-century survey, (ed. R.Herken) Springer 1995 (2nd ed.) [Introduces an interesting new notion related to "Kolmogorov complexity."]

[3] D.E.Knuth: Art of computer programming, Addison-Wesley.

[4] T.Cormen, C.Leiserson, R.Rivest: Introduction to Algorithms, MIT Press 1990.

[5] R.G. Gallager: Information Theory and Reliable Communication, Wiley, 2nd edition, 1980.

[6] M.R.Garey & D.Johnson: Computers and intractability: a guide to the theory of NP-completeness, W. H. Freeman, 1979; also updated in Johnson's "NP-completeness columns" in Journal of Algorithms. Also an article by Johnson in [7].

[7] Handbook of theoretical computer science vol A. (Elsevier).

[8] A.Heyting: Intuitionism, North-Holland, 1971. [Also, see the collected works of L.E.J.Brouwer for information about intuitionism. I find their arguments very appealing and think the general shunning of this area by most of the mathematical community, has been unjustified.]

[9] Hopcroft, J. E. & Ullman, J.D.: Introduction to automata theory, languages, and computation, Addison-Wesley, 1979. [One of the few textbooks discussing EXPTIME, pages 350-361.]

[10] K.Mehlhorn: Data structures and algorithms, Springer-Verlag, 1984.

[11] Minsky, Marvin Lee: Computation: finite and infinite machines, Englewood Cliffs, N.J., Prentice-Hall 1967, & reprinted. [Turing machines, counter machines, etc.]

[12] Robinson, R.M.: Minsky's small universal Turing machine, Int'l J. Math 2,5 (1991) 551-562. [Surveys small universal Turing machines.]