NOTES ON COMPLEXITY AND NP COMPLETENESS

BACKGROUND

P is the class of decision problems that are solvable by algorithms that run in polynomial time (as a function of input size).

```
That is, P = {q: q is a decision problem, and there exists
                 an algorithm A_q and constants c and k,
                 such that for all inputs i of size n,
                 A_q(i) terminates in time < cn^k and outputs
                 the correct answer (yes or no) to the decision
                 problem for that input}
```

Examples:

1. DECIDE-SORT(A1,A2) is in P; for there are many polynomial-time algorithms that can decide whether A2 is the sorted version of A1. For instance, we can simply run MergeSort on A1 and compare the result to A2. This takes n lg n time, which is less than n^2.

A variant on this, SORTED(A), asks whether A is already sorted. This can be solved in linear time: just pass through the array once, comparing each value to the next and terminating (with "no") if any pair is not in proper order.

2. IS-PRIME(n) appears to be in P. First of all, it seems "easy" to test whether an input n is prime: simply divide it by all the integers between 2 and n-1; then n is prime iff no remainders arise. This takes linear time, c(n-2), where c is the time to do one division. We can do even better: only integers between 2 and sqrt(n) need be examined.

Actually, there is a subtlety here: n is the (single) input value, rather than the "size" of the input. For instance, if n=100, then one can reasonably claim that the size of the input is 3 (digits), ie, 1 + log_10 (100), rather than 100. If we view things this way, then IS-PRIME(n) might take time that is nowhere close linear in the input size. Specifically, if something like sqrt(n) steps is the best we can do then this is exponential in the size s of the input: n = Theta(10^(2s-2)) where s = 1 + log_10 (n^0.5). (I have ignored floors here; it should actually be floor of log_10.)

Measuring input size this way, it is not clear whether IS-PRIME is in P. The above algorithm does not run that fast, but it is possible that a different algorithm might. The widespread suspicion is that IS-PRIME is in P, but this may turn out to be false. One bit of evidence is that there are known fast "approximate" algorithms that "almost always" give the right answer to IS-PRIME.

3. IS-COMPOSITE(n) also is thought to be in P: For if we can decide (yes or no) whether n is prime, that very same decision tells us whether n is composite, and in the same amount of time. More generally, the negation of a decision problem is another decision problem of the same complexity.

4. FACTOR(n,k) is another companion problem to IS-PRIME and IS-COMPOSITE: does n have a factor between 2 and k? But this is thought to be very hard, ie, not in P.

5. HALT is not in P; in fact HALT is unsolvable, period. (Recall HALT is the problem of deciding whether, for an input program p that takes no data, p eventually halts. Turing showed this is unsolvable: no algorithm can correctly produce the right answer (yes or no) for every input program p.)

Recalling ideas of order of growth, problems that are solvable by algorithms that run in any of the following (worst-case) runtimes are in P:

```
1/n     1     lg n     n     n lg n     n^k
```

whereas a problem that requires runtimes of more than n^k for all k are not in P. We regard the lower-bound worst-case runtime of a problem as a measure of its "complexity", ie how much "work" it takes to solve it.

PROBLEM VERIFICATION

So, some problems are not solvable at all, and some are solvable in polynomial time. What about in between cases, solvable but only VERY slowly? The class NP seems to be an excellent candidate for many such problems; seems to be -- for we are not certain. There is much evidence suggesting that NP contains many problems solvable too slowly to be in P, and we will review that evidence below. But first we need to define what NP is, and this requires first defining the notion of verification.

We already have the notion of a decision problem q being solved by an algorithm A(i). Now we introduce a weaker notion: that of q being "verified" by an algorithm V. We say q is verified by V if V takes two inputs, i (an input to q) and another one, y, where for every i there is a y such that V(i,y) = yes (or 1) iff the correct answer to q with input i is in fact yes.

Here is an example: the problem IS-COMPOSITE(n) [SN'T-PRIME(n)] might take a lot of work to check all possible divisors of n one by one; but suppose someone told you that n=p*q. You could simply multiply p and q and see if you get n, and if so then n is composite. That is, you can easily verify that the answer to IS-COMPOSITE(n) is yes, if given a suitable hint. That is the idea behind V; it is an algorithm that can take proper advantage of suitable hints.

Thus V verifies or checks that the answer is yes, with some "help" in the extra information given by a hint y. One can also think of y as "evidence" (some people prefer the word "certificate") for the answer being yes. A good analogy is that of proving a theorem: it can be hard to find a proof, but often easier to check (verify) someone else's proof.

For another analogy: it maybe hard to PROVE that two people are married, but if they show you their marriage certificate, all you have to do is verify it (eg by calling the relevant County Court office).

For a third analogy: one can imagine that V "guesses" a good i and then checks it. We will give a fourth analogy later.

Note that, unlike *solving* a decision problem outright (with no hints), verifying a hint for a problem is *not* the same as verifying a hint for its negation. Thus it is easy to see what a good hint is for IS-COMPOSITE (just give the two divisors). But what might a good hint be for IS-PRIME? Perhaps a *proof* that n is prime; but this might be very long, much longer than two divisors. So IS-COMPOSITE seems to be verifiable much more easily than IS-PRIME. [However, it was shown by Pratt that IS-PRIME is after all verifiable in polynomial time.] As for FACTOR, it is easily verified simply by the same sort of hint as for IS-COMPOSITE: just give the needed factor.

Again consider the ARE-MARRIED problem, and its negation AREN'T-MARRIED. There is an obvious verification for the former, but not so clear for the latter. Presumably one would have to call *all* County Courts.

Here are some further examples of verifications:

1. SAT is the problem of deciding for a given input formula i in propositional logic, whether i is satisfiable (ie, whether i can be true). SAT can be solved by constructing the truth table for i, and looking for a row that has overall value T. But this is slow, since the table has 2^n rows.

SAT is verified by the algorithm V_sat(i,y) -- where i is a formula in propositional logic -- which computes the y-th row in the truth table for i, and checks whether it is assigned the value True. Since a formula i is satisfiable iff it is True in some row, then all we need is to let y be a row where i is True and hand that y to V. So V-sat has little to do, it is the choice of y that does most of the work. A verification algorithm just verifies that the selection of y really works. To be fair, in the case of SAT, V has to compute the truth-value of i in that row, but this is easy, just a linear time process of working out such things as that the truth-value of T->F is F, and so on, each of which can be done in one or two steps.

2. 3-CNF-SAT is the problem of deciding for a given input formula i that is in 3-CNF form, whether it is satisfiable. 3-CNF-SAT is verified by the same algorithm as SAT. [A propositional formula is in 3-CNF form if it is a conjuction of formulas each of which looks like L, L_1 v L_2, or L_1 v L-2 v L_3, where the L_i are either letters or negated letters. Eg: P & (-Q v R) & (Q v -R v P) & -R.

3. 3-COLOR is the problem of deciding for a given graph whether its vertices can be colored with only the colors, in such a way that no two vertices that joined by a mutual edge have the same color. 3-color is verified by simply guessing a coloring and checking that it works. This can take up to n^2 time, since there may be O(n^2) edges. On the other hand, solving 3-COLOR seems to be very hard: the fastest known algorithms take exponential time -- basically building and examining a huge number of paths in the search for one that works.

4. HAM is the problem of deciding for a given graph G, whether it has a Hamiltonian cycle, ie a path that ends where it starts and includes each node exactly once (except the start/end node). The known algorithms for solving HAM construct and examine very many paths, looking for one that works; and these take exponential time.

HAM is verified by the algorithm V_ham(i,y) -- where i is a graph -- which examines the cycle y to see if it is Hamiltonian. If i has a Hamiltonian cycle then let y be such a cycle, and V_ham will output a yes. Again, the work is in finding y, but V does not do this. Who does? That doesn't matter -- verification is simply the process of verifying that a proposed answer works.

5. TSP, the "traveling salesman problem" -- perhaps the best-known of such problems. Here a salesman is given a map of cites, and roads (and distances) between them. The problem is to determine, for a given "cost" k, whether there is a "tour" (hamiltonian cycle) of total length no more than k. This is verified by being given such a cycle and just checking that it is what it is supposed to be. But, again, known algorithms to solve (rather than verify) TSP, take exponential time.

6. HALT, surprisingly, is also verifiable (though not quickly like the above examples). Let V_halt(i,y) be this: run (simulate) algorithm i for y steps and see if it has halted. If in fact i halts eventually, then there must be some number of steps by which it has halted; so the hint to look at that number, y, will work. We run the simulation that many steps and we see that it has halted. But this cannot be said to run in polynomial time.

EXERCISE: Explain why V_halt(i,y) cannot run in polynomial time.

EXERCISE: Can V_halt(i,y) be correctly said to run in *any* particular time T(n) where n is the size of the input algorithm i?

Yet another way to think of verification is the following: suppose that we run V many times, once for each possible y; sometimes V may report yes, and sometimes it may not, even when i is kept fixed. In that sense V is non-deterministic: its output can vary from one run to another. Of course, it is not truly non-deterministic: y is varying. But if we pretend that y is being randomly assigned somehow during the execution of V (rather than being input as data) then it can be thought of as an algorithm that does not always give the same answer even when its input (i) is kept fixed; and we can pretend that we get lucky and find a good y right away.

Now, as long as even *one* answer that V(i,y) outputs is yes, then q(i) is decided as true (by means of the random use of a suitable y). One can also imagine that instead of getting lucky and finding a good y by chance or guesswork, we run V for all y IN PARALLEL (say on many computers) and watch to see if any at all give a yes (ie have a good y).

Thus we can imagine, for instance, that we are checking all rows in the truth table for a proposition i (or all paths in a graph) at once (ie in parallel) and finding at least one of them that works (eg, has the truth value T, or is indeed a cycle of total length no more than k). This is the same as there being a verification algorithm for the problem.

Now, as we have seen, sometimes a verification algorithm for a problem q can be run fairly quickly, even if there is no known fast algorithm for solving q by itself. This motivates the following definition.

THE CLASS NP

The above examples indicate the potential interest of problems that, although they may be hard to solve by algorithms that run fast, they can be verified quickly. The following is a slightly loose definition (it leaves out one technical reuqirement, but will work for our purposes here):

```
NP = {q: q is a decision problem and there is a verification algorithm
         V for q, that runs in polynomial time as a function of the
         size n of the input i to q}
```

[The missing technical requirement is that the certificates for a given input i y are to have restricted lengths, namely O(polynomial(n)) where n is the length of i.]

Note: "NP" stands for "non-deterministic polynomial time". Probably VP ("verifiable in polynomial time") would have been a better name. [Also, the above is not quite the precise definition of NP; there is a subtlety about the size of the certificate y that should also be stipulated; but we will ignore this here.]

That is, NP consists of the (decision) problems that can be verified quickly.

It is clear that P is a subset of NP. For any problem q in P can be verified by V(i,y) = q(i); ie this V does us no good other than simply give us the original problem back (it ignores y) , but since q can be solved in polynomial time, then V gets its answer (yes or no) in that same time.

It is also clear that all the problems (except perhaps IS-PRIME) discussed above are in NP. [IS-PRIME is also, but this is not so obvious.]

The idea (and hope) is that even if some problem q might not be *decidable* in polynomial time, perhaps it can still be *verified* in polynomial time, and thus NP might be a larger class than P. But no one knows for sure. This is the famous unsolved conjecture P=NP.

There are several reasons that people tend to think that P and NP are not the same (ie that NP is larger). One is that, on intuitive grounds, it seems that being able to conclude q(i) from hints should allow one to decide genuinely tougher problems. Another is that we know of a number of problems (such as SAT) that are in NP (ie, that we can verify in polynomial time) and that we do not know how to decide in polynomial time. A third has to do with NP Completeness.

NP COMPLETENESS

The amazing fact about NP -- shown in 1971 by Cook -- is that it contains special problems, known as NP-complete problems, that are maximally difficult within NP. That is, an NP-complete problem p is one such that, roughly, if A_p solves p, then every q in NP can be solved by a variation on A_p that runs essentially as fast as A_p. More precisely, p is NP-complete if:

i. p is in NP
ii. all other problems q in NP can be "reduced in polynomial time" to solving p: there is a function f computable in polynomial-time such that for all inputs i to p, the correct answer to q(i) is yes iff the correct answer to p(f(i)) is yes.

Thus the time to answer q would be no more than that to answer p plus that time to compute f: to answer q(i) we would compute f(i) and then answer p(f(i)). Intuitively, the effort to solve p is (to within a polynomial amount of error) enough to solve any problem q in NP. This justifies the notion that p is maximally hard within NP.

It is not at all obvious that any problems are NP-complete. Cook showed SAT and 3-CNF-SAT are NP-complete. Later many other problems have been shown to be NP-complete as well. For instance, 3-COLOR is NP-complete.

The class of NP-complete problems is denoted NPC. Thus NPC is a subset of NP. And of course, P is a subset of NP. It is generally suspected that these are both *proper* subsets, ie that NP contains more than either P or NPC alone, but this is an open question. A picture that captures some of the intuitions is as follows:

```
      NP problems          even harder problems    unsolvable probs
 ----------------------    ----------------------  ------------------
  P              NPC
 ------  others  --------
 SORTED         HAM, SAT                                 HALT
```

That is, P and NPC are the two extremes within NP.

=================fanciful analogy, use at your own risk=================

We can perhaps overcome some of the abstractness of the definition of NPC with some help from comic-book superheroes. We suppose us ordinary people to be in P, that is of varying but seemingly limited powers. Then there are villians such as the Joker, Lex Luthor, etc, who appear much stronger than us, but not at the very top where Wonderwoman, Batman, Supeman, etc live (ie in NPC). The villians seem thus somewhere in between, in NP like eveyone else, but not at the bottom (P) nor the top (NPC) in power.

Thus every hero (every member of NPC) can outwit any of the villians and in particular can beat them at their own game; and the heros are all equivalent to each other in power except that their powers are of different sorts, eg Superman uses xray vision, Spiderman uses webs, etc. But they can all use their powers to achieve the same goals, and Batman for instance can stop the villian that Superman is after, it just might takes a little extra effort to use a web to do a job best suited for xray vision, etc. The key here is that it is only a *little* extra effort for one hero to do another's job: only polynomially more. And none of the villians can come anywhere near that close to the powers of the heros, the latter are all far more than polynomially beyond them.

=====end of fanciful analogy===

An important technique for showing a problem q to be NP-complete is to show:

i. q is in NP
ii. a known NP-complete problem p reduces to q

The second part shows that q is as hard as some NP-complete problem p, and thus that every problem in NP reduces to q (since every problem in NP reduces to p).

Example: It is known that 3-COLOR is NP-complete. Let us use this fact to show that 3-CNF-SAT is also NP-complete. We start by showing 3-COLOR reduces to 3-CNF-SAT. For let f be the function that computes as follows:

Let i be an input graph, and for each vertex x of i, form three propositional letters x_r, x_g, x_b (for red, green, blue). Then form the following axioms that correspond to the facts about the graph:

```
x_r or x_g or x_b       (each vertex is either red or green or blue)

x_r  ->  -x_g and -x_b
x_g  ->  -x_r and -x_b
x_b  ->  -x_r and -x_g  (no vertex can have more than one color)

x_b  -> -y_b
x_r  -> -y_r
x_g  -> -y_g   (for each vertex y connected to x by one edge)
```

Now, the above are easily rewritten as a conjunction in 3-CNF form (this is also part of what f does). This completes the computation of f; and it is a polynomial-time computation (in fact, linear time), in the size of the graph.

But the resulting 3-CNF formula is satisfiable iff the graph is 3-colorable: for simply take the letters, eg x_r, that are True and make those vertices the corresponding color (eg make x red), etc.

This shows that 3-COLOR reduces to 3-CNF-SAT, and given that 3-COLOR is NP-complete, it follows that also 3-CNF-SAT is NP-hard (ie, as hard as any problem in NP). Finally, since 3-CNF-SAT is also in NP, it is NP-complete. More concretely, with a solution to 3-CNF-SAT, we can construct a solution to any NP problem q: we first solve 3-COLOR using our solution to 3-CNF-SAT; this is what we did above with the complicated translation of graphs into formulas. And with the resulting solution to 3-COLOR we solve q (which we can do since 3-COLOR is in NPC). Altogether this takes the time for solving 3-CNF-SAT plus that to translate from 3-COLOR, plus that to translate from q; these translations are polynomial time, however.

EXERCISE: Does the above also show that 3-CNF-SAT reduces to 3-COLOR? Explain.

Of course, Cook already showed 3-CNF-SAT is NP-complete; the above simply illustrates the method most commonly used to show new problems are NP-complete.

Note: Both P and NPC are subsets of NP. It is widely believed, in large part due to the above results, that P and NPC are disjoint from each other. However, if even one problem in NPC is also in P, then NPC = P = NP. This latter follows from the very fact that NPC problems as as hard as any in NP, so if one NPC problem were solvable in PTIME, then all would be so, hence all would be in P.

The question as to whether in fact P = NP is the most famous open problem in all of computer science. The evidence in favor of it includes not only the rich theory discovered by Cook and others, but also the notion of verifiability: it seems intuitively implausible that the hidden work (the parallelism or guesswork) to find good "evidence" y for a polynomial time verification algorithm, can always be done in only a very little extra time.

EXERCISE: Prove that every problem in NP is solvable. [Hint: Use NPC.]