NP-Completeness

All of the algorithms we've studied so far have been polynomial-time algorithms: for an input size of n, their running time is $O(n^k)$ for some constant k. Recall from before that exponential functions grow much faster than polynomial functions and an algorithm with an exponential running time is useless except for a very small input size.

Problems with polynomial-time algorithms are called **tractable** while those requiring super-polynomial time are **intractable**.

One may wonder if a problem with an algorithm running in $\theta(n^{100})$ time, for example, is really tractable; it runs in polynomial time but it still will take a very long time. Polynomial-time algorithms of interest don't have exponents approaching 100; the highest exponent we've seen in all the algorithms we've studied in this course is 3. In practice, a problem with a polynomial-time algorithm is really tractable.

Here we examine some problems whose status is unknown; most computer scientists believe they are intractable but no one has proved that they are. They are called **NP-complete** problems.

A problem whose answer is simply either YES or NO is called a **decision problem**.

Consider the problem, "Given a weighted graph G = (V, E) and two vertices, u and v, in V what is the length of the shortest path from u to v?" This is not a decision problem because the answer is a number.

This problem can be converted to a decision problem, PATH(G, u, v, k): Given a weighted graph G = (V, E), two vertices, u and v, in V, and a nonnegative integer k, is there a path from u to v whose length is k or less?

One can find the length of the shortest path by solving PATH several times with different values of k.

Many abstract problems are optimization problems where some quantity is to be minimized or maximized. Such a problem can be reformulated into a decision problem in a manner like the shortest-path problem.

Complexity class P is the set of all decision problems that can be solved in polynomial time.

How should we measure the input size, n, for a decision problem? Count the number of items in the input like the length of an input array. The decision problem, “Is integer J a prime number?” has only one item, J, but it’s running time depends strongly on the value of J. The best measure of input size is the number of bits required to state the input.

PATH is a decision problem in complexity class P; we can use Dijkstra to find the length of the shortest path in O(|E| lg |V|) time and then compare this length to k to see if the answer to PATH is YES or NO.

Consider another graph problem. A Hamiltonian cycle in an undirected graph is a simple cycle that contains all the vertices in the graph. Let HAM-CYCLE(G) be the decision problem: Does G have a Hamiltonian cycle? If the answer to HAM-CYCLE(G) is YES then there is some permutation of the vertices in V[G], say $\{v_1, v_2, \ldots, v_{|V|}\}$ where G.E contains the edges $v_1 v_2$, $v_2 v_3$, …, $v_{|V|-1} v_{|V|}$, and $v_{|V|} v_1$. If the answer to HAM-CYCLE(G) is NO then there is no such permutation.

One way to solve HAM-CYCLE(G) is to try all |V|! permutations of the vertices and for each permutation examine G.E to see if it contains the required edges. If G is represented with adjacency lists each permutation can be tested in O(|V| + |E|) time, a polynomial time.

Unfortunately there are |V|! permutations and the factorial function grows faster than any polynomial. If the answer to HAM-CYCLE(G) is NO this algorithm will require Ω(|V|!) time to find the answer. If the answer to HAM_CYCLE(G) is YES the running time depends on luck; how many permutations does the algorithm try before it finds one that’s a Hamiltonian cycle.

Suppose a friend says that G has a Hamiltonian cycle and gives us a permutation of the vertices. We could check the given permutation to see if it's a Hamiltonian cycle in polynomial time. HAM-CYCLE is a decision problem where any one proposed solution can be verified in polynomial time.

A decision problem where any one proposed solution can be verified in polynomial time is said to be in **complexity class NP**. One member of class NP is HAM-CYCLE.

Any decision problem in class P is also in class NP. For example, PATH is in NP; if a friend gives us a proposed path from vertex u to vertex v we can ignore the proposed path and just run Dijkstra in polynomial time.

There are many important problems in class NP; some of them, like PATH, are in class P. Many of the problems in NP, like HAM-CYCLE, have an unknown status. No one knows if HAM-CYCLE is in class P or not; nobody has found a polynomial-time algorithm to solve HAM-CYCLE but nobody has proved that no such algorithm can exist.

Problem reduction is a useful tool, to solve problem Q for some given input:

1. transform the input into the input for another problem, Q* (the transformation should handle any input Q),
2. solve Q*, and
3. use the output of Q* to generate the output of Q.

If Q and Q* are both decision problems then the third step can be eliminated as long as the answer generated by Q* (YES or NO) is always the correct answer for problem Q.

The running time is the sum of the times for each step. The reduction isn't very helpful if step (1) takes a very long time; at the very least we should insist that step (1) runs in polynomial time.

Decision problem Q is said to be reducible to decision problem Q* if there exists a transformation, T, so that:

a) if X is any input to Q then T(X) is an input to Q*

b) T runs in polynomial time, and

c) the answer (YES or NO) of problem Q* given input T(X) is always the same as the answer of problem Q given input X.

Let decision problem Q be reducible to decision problem Q*. What does the complexity of one of these problems say about the complexity of the other problem?

a) If Q* is in class P, then Q is also in class P (reduce Q to Q* in polynomial time and then solve Q* in polynomial time).

b) If Q is not in class P, then Q* can't be in class P either (otherwise the reduction would put Q in class P).

If Q is in class P doesn't that put Q* in class P? No. The reduction may have reduced an "easy" problem to a "hard" problem. Similarly, if Q* is not in class P, Q may be in class P or not.

We now examine another decision problem, CIRCUIT-SAT. Computers are built with Boolean logic elements like AND-gates, OR-gates, and NOT-gates. Each gate receives a number of inputs and produces one output. Each input and output is a variable with two possible values, 0 or 1. The output of an AND-gate equals 1 if and only if all its inputs are equal to 1. The output of an OR-gate equals to 0 if and only if all its inputs are equal to 0. A NOT-gate has one input; its output is 1 if and only if its input is equal to 0.

A combinational circuit is a set of gates connected together with no cycles (its directed graph is acyclic) with one output and n inputs. The output of a combinational circuit is a Boolean function of its n inputs. There are 2^n ways of assigning 0 or 1 values to the inputs and each input assignment produces a 0 or a 1 value on the output. We can think of a combinational circuit as looking for certain input assignments and producing a 1-value on its output if and only if one of those input assignments is fed into the circuit. The circuit is satisfied whenever such an assignment is fed in.

The CIRCUIT-SAT decision problem is : “Given a combinational circuit of AND-gates, OR-gates, and NOT-gates is there an input assignment that satisfies it?”

Any given input assignment can be checked in polynomial time so CIRCUIT-SAT is in class NP.

One way to solve CIRCUIT_SAT on a circuit with n inputs is to exhaustively check all possible 2^n input assignments to see if any produces a 1-output. This algorithm requires superpolynomial time. No one has found a polynomial time algorithm to solve CIRCUIT-SAT.

CIRCUIT_SAT is a very interesting decision problem because any problem in class NP can be reduced to it!

The proof of this statement has a lot of details but the basic idea behind it is simple. If problem X is any decision problem in class NP then it has a polynomial time algorithm, V, to verify a proposed solution to the problem. For input size n, the number of steps V requires on a computer is in $O(n^k)$ for some constant k.

One can design a combinational circuit, C, to simulate V for input size n; the inputs to C will be the bits of the input to V and the single output of C will equal to 1 if and only if the output of V is YES. This transforms problem X to CIRCUIT-SAT.

What does this mean? If somebody finds a polynomial time solution to CIRCUIT-SAT putting CIRCIT-SAT in class P, then all decision problems in class NP have polynomial time solutions and are in class P.

But nobody has found such a solution to CIRCUIT-SAT. Most researchers believe that a polynomial time doesn’t exist; class NP has hundreds of problems with no known polynomial time solutions so it’s very doubtful that they all could be solved so simply.

A decision problem, Q*, where any problem in class NP can be reduced to Q* is called **NP-hard**. If Q* itself is in class NP then Q* is called **NP-complete**. CIRCUIT-SAT in NP-complete.

Many other decision problems are also NP-complete. To prove that some problem, Q*, in class NP is also NP-complete do we have to show that all problems in class NP are reducible to Q*? Not really; all we need to show that some problem known to be NP-complete (like CIRCUIT-SAT) is reducible to Q*.

Hundreds of decision problems are known to be NP-complete. They are in a wide variety of domains: Boolean logic, graphs; arithmetic, network design, sets and partitions, storage and retrieval, sequencing and scheduling, mathematical programming, algebra and number theory, games and puzzles, automata and language theory, program optimization, etc.

It's like the proverbial problem of trying to find a needle in a haystack; we have a haystack of NP-complete problems and the needle we're looking for is an NP-complete problem with a polynomial time algorithm. But it's very doubtful that the needle really exists and meanwhile more and more hay is added to the stack!

Many optimization problems are NP-complete. The only known algorithm to find the optimum solution for such a problem is superpolynomial. In some cases a near-optimal solution can be found with a polynomial time algorithm.