Complexity

So far in 6.006, we've focused on explicit runtimes of algorithms: $O(n), O(\log n), O(n^2)$, etc. In the theory of computing, rather than talking about runtimes, we often refer to a problem or an algorithm's "complexity class". The major complexity classes we'll talk about are:

P: problems solvable in **polynomial** (n^c) time, where n is the size of the input and c is some constant independent of n.
NP: decision problems whose solutions can be verified in **polynomial** time.
EXP: problems solvable in **exponential**, 2^{n^c}, time.
R: problems solvable in **finite** time.

Important relationship: $P \subset EXP \subset R$
Important relationship: $P \subseteq NP$
Important open-problem: $P \neq NP$, believed to be true.

Decision problems: problems in which you are given some input and have to return either true or false. For example, "is there a path between s and t in graph G?", or "is there a satisfiable assignment to the variables a, b, c such that $(\overline{a \cup b} \cap (c \cup \overline{a})$?"

When we say that a *problem's* complexity is in X, we mean that there is exists algorithm that correctly solves that problem and that has a runtime $\in X$.

Unsolvable problems

Some problems are unsolvable - meaning that no matter how much time you're given to try to solve them, there is no algorithm that runs in a finite amount of time that would always return the correct answer. As an example:
The Halting Problem: Given a computer program, does it ever halt (meaning it never goes into an infinite loop)? (YES or NO **decision problem**). The halting problem is **uncomputable**: it is solvable in infinite time because you can just wait for the input to terminate (or halt), but no program can solve this problem in finite time.

In fact, it turns out lots and lots of problems are unsolvable! Really coarsely, this stems from the fact that there are much more problems than possible algorithms. More specifically, each function $f : \{0,1\}^n \to \{0,1\}$ (called a boolean function) describes a different decision problem by specifying the desired output (either 0 or 1) for every possible input binary string. Since the set of those functions is uncountable (recall the diagonalization argument from 6.042), the set of decision problems is uncountable. Now, every algorithm can be described by a binary string (for example the binary representation of your Python code that implements that algorithm). Therefore, since the set of binary strings is countable, the set of algorithms is also countable. Therefore, there is no bijection from decision problems to algorithms. So, there is not an algorithm for every problem. Note: **that explanation is very handwavy, and you are definitely not accountable for understanding it.** For more context (after finals week?), check out this link: `https://concretenonsense.`

wordpress.com/2011/07/12/cantors-diagonal-argument-and-undecidability/

P - polynomial time

The primary complexity class we've dealt with so far this semester is "P" - the class of all problems that are decidable in polynomial time. P includes problems like "sort an unsorted list of integers" (which takes $O(n \log n)$), or "find the shortest path in a positive-weighted, directed graph G = (V,E)" (which takes $O(|E| + |V| \log |V|)$. In fact, P covers any problem that is solvable by an algorithm that takes $O(n^c)$ time.

NP - nondeterministic polynomial time

Problems in NP include any problem that is *verifiable* in polynomial time. This means that while finding a solution itself might be difficult, if you tell me what you claim is a solution, I can check whether you're right in polynomial time.

SAT - boolean satisfiability problem

As an example: SAT, or the "boolean satisfiability problem", is the problem of determining whether there is a valid assignment to the literals $\Sigma = a, b, c, \ldots$ such that some boolean expression $S = ((a \cap b) \cup (a \cap c \cap \overline{b})) \cap (\overline{a} \cup \overline{b} \cup \overline{c}) \cap \ldots$ is true. Specifically, the input to the problem is a boolean formula, and the output to the problem is whether it is satisfiable (we don't have to find a satisfying assignment). Clearly, if you show me an answer, meaning an assignment "$a = True, b = False, c = True, \ldots$", I can determine whether your answer is correct in linear time by just plugging in your values into S and evaluating the boolean formula to check if S is now true.

We can figure out whether an assignment of the variables in a SAT problem causes the expression to be satisfied in polynomial time. However, it's less clear how we would *find* a valid assignment if we didn't know one. One method would be to try every possible assignment to the variables: "a=False, b = False, c=False, ...", then "a = True, b = False, c=False, ..." and so on. However, doing all of these computations sequentially would take a while: there are $2^{\#\text{ variables}}$ such configurations, so this procedure would take *exponential* time, i.e. $\in$ EXP.

It would be nice if we could somehow find a polynomial time algorithm to quickly find whether a correct assignment to the variables existed. However, we don't know if one exists, i.e. we aren't sure if SAT $\in$ P.

It turns out that if there was such an algorithm, then it would imply that P = NP. Note that that's not inherently obvious: for example, we do know that checking if there's a path from s to t in a graph G takes $O(|E| + |V|) = O(n)$ is in P and thus also in NP, but that unfortunately that doesn't tell us anything about whether P = NP. It turns out that the SAT problem in particular has a special property, called NP-completeness, which guarantees that if SAT is in P, then every problem in NP is also in P.

P vs. NP

A great deal of time has been spent trying to determine whether problems are in P (easily solvable), NP (likely not easily solvable), or EXP (definitely unsolvable). Because we aren't sure exactly how these classes relate (beyond $P \subseteq NP \subset EXP$), we've created some more complexity classes to describe the relative complexity of problems:

NP-hard: problems that are as hard as any problem in NP.
NP-complete: problems that are in NP and are NP-hard.
EXP-hard: problems that are as hard as any problem in EXP.
EXP-complete: problems that are in EXP and are EXP-hard.

The million dollar question of the century is whether P equals NP.

Reductions

Reductions are a way of *delegating* problem solving. We say that a problem A is *reducible* to problem B if we can find a way to solve problem A by transforming it into problem B and solving problem B instead. For instance, we can solve the "kth largest element in array" problem (problem A) by reducing it to sorting the array (problem B) and returning the kth element.

We can use $A \rightarrow B$ to denote that there exists a reduction from problem A to problem B.

The goal of a reduction is to convert your problem A into a problem B that you already know how to solve. After turning your problem A into the other problem B, you can infer something about the *hardness* of problem B given that you know the hardness of problem A.

Suppose you knew that A is a really hard problem (maybe it is $\Omega(2^n)$-time) and you've reduced A to another problem B. If B were a very easy problem that you can solve in $poly(n)$ time, and the reduction itself $A \rightarrow B$ (transforming an input for A into an input for B) also took you $poly(n)$ time, then how fast could you solve A now?

If this reduction existed, then you could solve A in $poly(n)$ time by reducing A to B in $poly(n)$ time and then solving B in $poly(poly(n))$ time. This gives you a contradiction to the assumption that A is $\Omega(2^n)$. Thus, you can never reduce A to such a problem B.

This implies that any problem B that you can reduce A to will have to be **at least as hard as A**.

Most important idea: You can use a reduction $A \rightarrow B$ to prove that B is a very hard problem. For instance, people proved that $LONGPATH$ is a very hard problem by reducing $HAMPATH$ to $LONGPATH$.

Reduction examples

"hamiltonian path" $\rightarrow$ "longest simple path in a graph"

A Hamiltonian path from s to t in a graph G is a path from s to t that visits all the other vertices in the graph *exactly* once. The $HAMPATH$ decision problem is given $\langle G, s, t\rangle$ and returns yes/no to indicate if a Hamiltonian path exists from s to t in G. The $LONGPATH$ decision problem is given $\langle G, s, t, k\rangle$ and returns yes if there exists a path from s to t in G of length at least k edges or no otherwise. How can we solve $HAMPATH$ using $LONGPATH$ and thus prove that $LONGPATH$ is as hard as $HAMPATH$ (or harder)? We can notice that a Hamiltonian path will have exactly $|V| - 1$ edges since it visits all the vertices in the graph once. Also notice that there's no simple path longer than $|V| - 1$ edges in any graph. If there were, then a node would have to be repeated and it would not be a simple path. So it seems that we can solve $HAMPATH$ on $\langle G, s, t\rangle$ by using $LONGPATH$ on $\langle G, s, t, |V| - 1\rangle$. That's our reduction that gives us the answer to $HAMPATH$. This shows that $LONGPATH$ must be as hard as $HAMPATH$ and since $HAMPATH$ is NP-complete, that makes $LONGPATH$ pretty darn hard.

"unweighted shortest path" $\rightarrow$ "shortest weighted path with no negative edges"

Assume we don't know how to solve "unweighted shortest path," but we know how to solve "shortest weighted path with no negative edges" with Dijkstra's algorithm. We also know that Dijkstra's algorithm $\in P$. We can transform the unweighted graph by making all edges weight 1. From this reduction, we know that finding "shortest weighted path with no negative edges" is at least as hard as "unweighted shortest path" so the problem of finding an "unweighted shortest path" $\in P$.

Hamiltonian path $\rightarrow$ Hamiltonian cycle

Given that you know Hamiltonian path is NP-complete, then you can add a node to the graph input into the $HAMPATH$ problem and connect that node with every other node in the graph. Then, the transformed input into Hamiltonian cycle will return "True" if and only if the original input to Hamiltonian path returns "True." Thus, "Hamiltonian cycle" is NP-hard.

You can further prove that a problem is NP-complete after it has been proven to be NP-hard by showing that it is in the set NP.