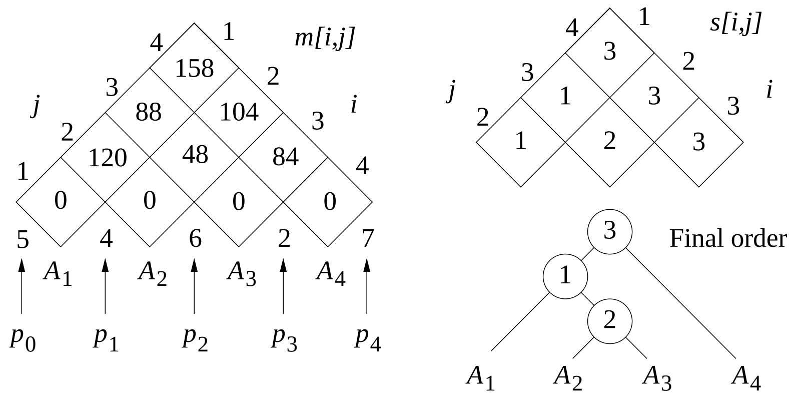

Figure 34: Chain Matrix Multiplication.

In the figure below we show an example. This algorithm is tricky, so it would be a good idea to trace through this example (and the one given in the text). The initial set of dimensions are $\langle 5, 4, 6, 2, 7\rangle$ meaning that we are multiplying A_1 (5×4) times A_2 (4×6) times A_3 (6×2) times A_4 (2×7). The optimal sequence is $((A_1(A_2A_3))A_4)$.

Lecture 27: NP-Completeness: General Introduction

(Tuesday, May 5, 1998)

Read: Chapt 36, up through section 36.4.

Easy and Hard Problems: At this point of the semester hopefully you have learned a few things of what it means for an algorithm to be efficient, and how to design algorithms and determine their efficiency asymptotically. All of this is fine if it helps you discover an acceptably efficient algorithm to solve your problem. The question that often arises in practice is that you have tried every trick in the book, and still your best algorithm is not fast enough. Although your algorithm can solve small problems reasonably efficiently (e.g. $n \leq 20$) the really large applications that you want to solve (e.g. $n \geq 100$) your algorithm does not terminate quickly enough. When you analyze its running time, you realize that it is running in *exponential time*, perhaps $n^{\sqrt{n}}$, or 2^n, or $2^{(2^n)}$, or $n!$, or worse.

Towards the end of the 60's and in the eary 70's there were great strides made in finding efficient solutions to many combinatorial problems. But at the same time there was also a growing list of problems for which there seemed to be no known efficient algorithmic solutions. The best solutions known for these problems required exponential time. People began to wonder whether there was some unknown paradigm that would lead to a solution to these problems, or perhaps some proof that these problems are inherently hard to solve and no algorithmic solutions exist that run under exponential time.

Around this time a remarkable discovery was made. It turns out that many of these "hard" problems were interrelated in the sense that if you could solve any one of them in polynomial time, then you could solve all of them in polynomial time. The next couple of lectures we will discuss some of these problems and introduce the notion of P, NP, and NP-completeness.

Polynomial Time: We need some way to separate the class of efficiently solvable problems from inefficiently solvable problems. We will do this by considering problems that can be solved in polynomial time.

We have measured the running time of algorithms using worst-case complexity, as a function of n, the size of the input. We have defined input size variously for different problems, but the bottom line is the number of bits (or bytes) that it takes to represent the input using any *reasonably efficient encoding*. (By a reasonably efficient encoding, we assume that there is not some significantly shorter way of providing the same information. For example, you could write numbers in unary notation $11111111_1 = 100_2 = 8$ rather than binary, but that would be unacceptably inefficient.)

We have also assumed that operations on numbers can be performed in constant time. From now on, we should be more careful and assume that arithmetic operations require at least as much time as there are bits of precision in the numbers being stored.

Up until now all the algorithms we have seen have had the property that their worst-case running times are bounded above by some *polynomial* in the input size, n. A *polynomial time algorithm* is any algorithm that runs in time $O(n^k)$ where k is some constant that is independent of n. A problem is said to be *solvable in polynomial time* if there is a polynomial time algorithm that solves it.

Some functions that do not "look" like polynomials but are. For example, a running time of $O(n \log n)$ does not look like a polynomial, but it is bounded above by a the polynomial $O(n^2)$, so it is considered to be in polynomial time.

On the other hand, some functions that do "look" like polynomials are not. For example, a running time of $O(n^k)$ is *not* considered in polynomial time if k is an input parameter that could vary as a function of n. The important constraint is that the exponent in a polynomial function must be a *constant* that is independent of n.

Decision Problems: Many of the problems that we have discussed involve *optimization* of one form or another: find the shortest path, find the minimum cost spanning tree, find the knapsack packing of greatest value. For rather technical reasons, most NP-complete problems that we will discuss will be phrased as decision problems. A problem is called a *decision problem* if its output is a simple "yes" or "no" (or you may think of this as True/False, 0/1, accept/reject).

We will phrase many optimization problems in terms of decision problems. For example, rather than asking, what is the minimum number of colors needed to color a graph, instead we would phrase this as a decision problem: Given a graph G and an integer k, is it possible to color G with k colors. Of course, if you could answer this decision problem, then you could determine the minimum number of colors by trying all possible values of k (or if you were more clever, you would do a binary search on k).

One historical artifact of NP-completeness is that problems are stated in terms of *language-recognition problems*. This is because the theory of NP-completeness grew out of automata and formal language theory. We will not be taking this approach, but you should be aware that if you look in the book, it will often describe NP-complete problems as languages.

Definition: Define *P* to be the set of all decision problems that can be solved in polynomial time.

NP and Polynomial Time Verification: Before talking about the class of NP-complete problems, it is important to introduce the notion of a verification algorithm. Many language recognition problems that may be very hard to solve, but they have the property that it is easy to *verify* whether its answer is correct.

Consider the following problem, called the *undirected Hamiltonian cycle problem* (UHC). Given an undirected graph G, does G have a cycle that visits every vertex exactly once.

An interesting aspect of this problems is that *if* the graph did contain a Hamiltonian cycle, then it would be easy for someone to *convince* you that it did. They would simply say "the cycle is $\langle v_3, v_7, v_1, \ldots, v_{13}\rangle$". We could then inspect the graph, and check that this is indeed a legal cycle and that it visits all the vertices of the graph exactly once. Thus, even though we know of no efficient

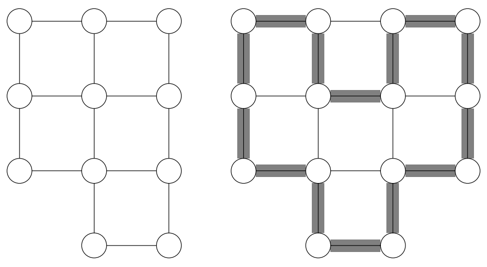

Figure 35: Undirected Hamiltonian cycle.

way to *solve* the Hamiltonian cycle problem, there is a very efficient way to *verify* that a given graph is Hamiltonian. The given cycle is called a *certificate*. This is some piece of information which allows us to verify that a given string is in a language. If it is possible to verify the accuracy of a certificate for a problem in polynomial time , we say that the problem is *polynomial time verifiable*.

Note that not all languages have the property that they are easy to verify. For example, consider the problem of determining whether a graph has *exactly one* Hamiltonian cycle. It would be easy for someone to convince that it has at least one, but it is not clear what someone (no matter how smart) would say to you to convince you that there is not another one.

Definition: Define *NP* to be the set of all decision problems that can be verified by a polynomial time algorithm.

Beware that polynomial time verification and polynomial time solvable are two very different concepts. The Hamiltonian cycle problem is NP-complete, and so it is widely believed that there is no polynomial time solution to the problem.

Why is the set called "NP" rather than "VP"? The original term NP stood for "nondeterministic polynomial time". This referred to a program running on a *nondeterministic computer* that can make guesses. Basically, such a computer could nondeterministically guess the value of certificate, and then verify that the string is in the language in polynomial time. We have avoided introducing nondeterminism here. It would be covered in a course on complexity theory or formal language theory.

Like P, NP is a set of languages based on some complexity measure (the complexity of verification). Observe that P $\subseteq$ NP. In other words, if we can solve a problem in polynomial time, then we can certainly verify that an answer is correct in polynomial time. (More formally, we do not even need to see a certificate to solve the problem, we can solve it in polynomial time anyway).

However it is not known whether P $=$ NP. It seems unreasonable to think that this should be so. In other words, just being able to verify that you have a correct solution does not help you in finding the actual solution very much. Most experts believe that P $\neq$ NP, but no one has a proof of this.

NP-Completeness: We will not give a formal definition of NP-completeness. (This is covered in the text, and higher level courses such as 451). For now, think of the set of *NP-complete* problems as the "hardest" problems to solve in the entire class NP. There may be even harder problems to solve that are not in the class NP. These are called *NP-hard* problems.

One question is how can we the notion of "hardness" mathematically formal. This is where the concept of a reduction comes in. We will describe this next.

Reductions: Before discussing reductions, let us first consider the following example. Suppose that there are two problems, A and B. You know (or you strongly believe at least) that it is impossible to solve

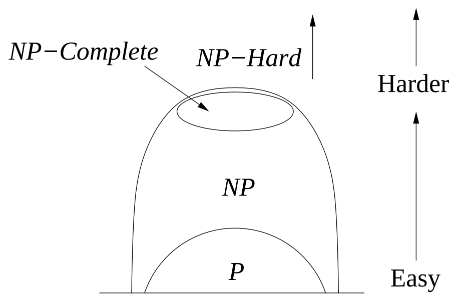

Figure 36: Relationship between P, NP, and NP-complete.

problem A in polynomial time. You want to prove that B cannot be solved in polynomial time. How would you do this?

We want to show that

$$(A \notin \text{P}) \Rightarrow (B \notin \text{P}).$$

To do this, we could prove the contrapositive,

$$(B \in \text{P}) \Rightarrow (A \in \text{P}).$$

In other words, to show that B is not solvable in polynomial time, we will suppose that there is an algorithm that solves B in polynomial time, and then derive a contradiction by showing that A can be solved in polynomial time.

How do we do this? Suppose that we have a subroutine that can solve any instance of problem B in polynomial time. Then all we need to do is to show that we can use this subroutine to solve problem A in polynomial time. Thus we have “reduced” problem A to problem B.

It is important to note here that this supposed subroutine is really a *fantasy*. We know (or strongly believe) that A cannot be solved in polynomial time, thus we are essentially proving that the subroutine cannot exist, implying that B cannot be solved in polynomial time.

Let us consider an example to make this clearer. It is a fact that the problem of determining whether an undirected graph has a Hamiltonian cycle (UHC) is an NP-complete problem. Thus, there is no known polynomial time algorithm, and in fact experts widely believe that no such polynomial time algorithm exists.

Suppose your boss of yours tells you that he wants you to find a polynomial solution to a different problem, namely the problem of finding a Hamiltonian cycle in a *directed graph* (DHC). You think about this for a few minutes, and you convince yourself that this is not a reasonable request. After all, would allowing directions on the edges make this problem any easier? Suppose you and your boss both agree that the UHC problem (for undirected graphs) is NP-complete, and so it would be unreasonable for him to expect you to solve this problem. But he tells you that the directed version is easier. After all, by adding directions to the edges you eliminate the ambiguity of which direction to travel along each edge. Shouldn’t that make the problem easier? The problem is, how do you convince your boss that he is making an unreasonable request (assuming your boss is willing to listen to logic).

You explain to your boss: “Suppose I could find an efficient (i.e., polynomial time) solution to the DHC problem, then I’ll show you that it would then be possible to solve UHC in polynomial time.” In particular, you will use the efficient algorithm for DHC (which you still haven’t written) as a subroutine to solve UHC. Since you both agree that UHC is not efficiently solvable, this means that this efficient subroutine for DHC must not exist. Therefore your boss agrees that he has given you an unreasonable task.

Here is how you might do this. Given an undirected graph G, create a directed graph G' by just replacing each undirected edge $\{u, v\}$ with two directed edges, (u, v) and (v, u). Now, every simple path in the G is a simple path in G', and vice versa. Therefore, G has a Hamiltonian cycle if and only if G' does. Now, if you could develop an efficient solution to the DHC problem, you could use this algorithm and this little transformation solve the UHC problem. Here is your algorithm for solving the undirected Hamiltonian cycle problem. You take the undirected graph G, convert it to an equivalent directed graph G' (by edge-doubling), and then call your (supposed) algorithm for directed Hamiltonian cycles. Whatever answer this algorithm gives, you return as the answer for the Hamiltonian cycle.

UHC to DHC Reduction

```
bool Undir_Ham_Cycle(graph G) {
    create digraph G' with the same number of vertices as G
    for each edge {u,v} in G {
        add edges (u,v) and (v,u) to G'
    }
    return Dir_Ham_Cycle(G')
}
```

You would now have a polynomial time algorithm for UHC. Since you and your boss both agree that this is not going to happen soon, he agrees to let you off.

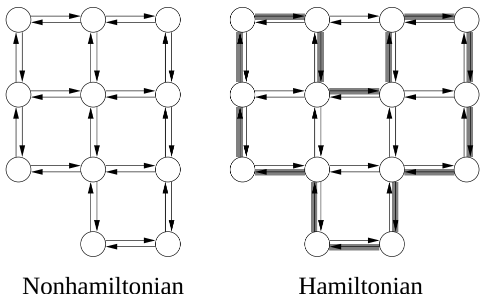

Figure 37: Directed Hamiltonian cycle reduction.

Notice that neither problem UHC or DHC has been solved. You have just shown how to convert a solution to DHC into a solution for UHC. This is called a *reduction* and is central to NP-completeness.

Lecture 28: NP-Completeness and Reductions

(Thursday, May 7, 1998)

Read: Chapt 36, through Section 36.4.

Summary: Last time we introduced a number of concepts, on the way to defining NP-completeness. In particular, the following concepts are important.

Decision Problems: are problems for which the answer is either “yes” or “no.” The classes P and NP problems are defined as classes of decision problems.

P: is the class of all decisions problems that can be solved in polynomial time (that is, $O(n^k)$ for some constant k).