MIT OpenCourseWare
http://ocw.mit.edu

6.080 / 6.089 Great Ideas in Theoretical Computer Science
Spring 2008

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.

6.080/6.089 GITCS	March 4, 2008
Lecture 8	
Lecturer: Scott Aaronson	*Scribe: Hristo Paskov*

1 Administrivia

There will be two scribes notes per person for the term. In addition, you may now use the book to review anything not clear from lecture since we're on complexity.

2 Recap

In most cases of interest to us, the real question is not what's computable; it's what's computable with a reasonable amount of time and other resources. Almost every problem in science and industry is computable, but not all are *efficiently* computable. And even when we come to more speculative matters – like whether it's possible to build a machine that passes the Turing Test – we seethat with unlimited resources, it's possible almost trivially (just create a giant lookup table). The real question is, can we build a thinking machine that operates within the time and space constraints of the physical universe? This is part of our motivation for switching focus from computability to complexity.

2.1 Early Results

2.1.1 Claude Shannon's counting argument

Most Boolean functions require enormous circuits to compute them, i.e. the number of gates is exponential in the number of inputs. The bizarre thing is we know this, yet we still don't have a good example of a function that has this property.

2.1.2 Time Hierarchy Theorem

Is it possible that everything that is computable at all is computable in linear time? No. There is so much that we don't know about the limits of feasible computation, so we must savor what we do know. There are more problems that we can solve in n^3 than in n^2 steps. Similarly, there are more problems that we can solve in 3^n than in 2^n steps. The reason this is true is that we can consider a problem like:
"Given a Turing Machine, does it halt in $\leq n^3$ steps?"
Supposing it were solvable in fewer than n^3 steps, we could take the program that would solve the problem and feed it itself as input to create a contradiction. This is the finite analog of the halting problem.

2.1.3 Existence of a fastest algorithm

One thing we have not mentioned is a bizarre phenomenon in runtime people discovered in the 60's:

Does there have to be a fastest algorithm for every problem? Or, on the contrary, could there be an infinite sequence of algorithms to solve a problem, each faster, but no fastest one?

2.1.4 Concrete example: Matrix Multiplication

Given two $n \times n$ matrices, find:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \dots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \vdots & \vdots & \dots & \vdots \\ c_{n1} & c_{n2} & \dots & c_{nn} \end{pmatrix}$$

2.1.5 Straightforward way

The straightforward way takes n^3 steps because of the way we multiply columns and rows. However, there do exist better algorithms.

2.1.6 Strassen's algorithm

In 1968, Strassen found an algorithm that takes only $O(n^{2.78})$ steps. His algorithm used a divide and conquer approach, repeatedly dividing the original matrix into $\frac{n}{2} \times \frac{n}{2}$ matricies and combining in clever way. It's a somewhat practical algorithm; people actually use it in scientific computing and other areas. Strassen's algorithm served as one of the early inspirations for the theory of efficient algorithms: after all, if people had missed something so basic for more than a century, then what else might be lurking in algorithm-land?

2.1.7 Faster and faster algorithms

Sometime later an algorithm that takes $O(n^{2.55})$ time was found. The best currently known algorithm is due to Coppersmith and Winograd, and takes $O(n^{2.376})$ algorithm. Many people believe that we should be able to do better. The natural limit is n^2 since, in the best case, we have to look at all entries of the matrices. Some people conjecture that for all $\epsilon > 0$, there exists an algorithm that takes $O(n^{2+\epsilon})$ time.

2.1.8 Practical considerations

If matrices are reasonably small, then you're better off multiplying them with the naïve $O(n^3)$ algorithm. It's an empirical fact that, as you go to asymptotically faster algorithms, the constant prefactor seems to get larger and larger. You *would* want to switch to a more sophisticated algorithm with larger matrices, since there the constant prefactor constant doesn't matter as much. For all we know, it could be that as you go to bigger and bigger matrices, there's an unending sequence of algorithms that come into play. Or the sequence could terminate; we don't know yet.

Note that we can't obviously just make an algorithm that chooses the best algorithm for a given matrix size, since each new faster algorithm might require some extra insight to come up with. If there *were* an algorithm to produce these algorithms, then we'd have a single algorithm after all.

This illustrates the notion of an infinite progression of algorithms with no best one. Note that this sequence is countable since there's only a countable number of algorithms.

2.1.9 Blum Speedup Theorem

In 1967, Blum showed that we can construct problems for which there's *provably* no best algorithm. Indeed, there exists a problem such that for every $O(f(n))$ algorithm, there's also an $O(log(f(n)))$

algorithm! How could this be? The problem would take a lot of time to solve by any algorithm—some giant stack of exponentials such that taking *log* any number of times won't get it down to a constant. We're not going to prove this theorem, but be aware that it exists.

2.1.10 Summary

So for every problem there won't necessarily be a best algorithm. We don't know how often this weird phenomenon arises, but we do know that it can in principle occur.

Incidentally, there's a lesson to be learned from the story of matrix multiplication. It was intuitively obvious to people that n^3 was the best we could do, and then we came up with algorithms that beat this bound. This illustrates why it's so hard to prove *lower* bounds: because you have to rule out every possible way of being clever, and there are many non-obvious ways.

3 The meaning of efficient

We've been talking about "efficient" algorithms, but what exactly do we mean by that?

One definition computer scientists find useful (though not the only one) is that "efficient" = "polynomial time." That is, an algorithm is efficient if there exists a k such that all instances of size n are solved in $O(n^k)$ time. Things like $\sqrt{n}$ or $logn$ are not polynomial, but there is a polynomial larger than them, hence they're also efficient.

No sooner do you postulate this criterion for efficiency than people think of objections to it. For example, an $n^{10,000}$-time algorithm is polynomial-time, but obviously not efficient in practice. On the other hand, what about $O(1.00000000001^n)$ which is exponential, but appears to be fine in practice for reasonably-sized n?

Theoretical computer scientists are well aware of these issues, and have two main responses:

1. Empirical - in practice efficient usually does translate into polynomial time; inefficient usually translates into exponential time and worse with surprising regularity. This provides the empirical backbone of theory. Who invents $O(n^{10,000})$ algorithms?

2. Subtle response - any criterion for efficiency has to meet the needs of practitioners *and* theorists. It has to be convenient. Imagine a subroutine that takes polynomial time and an algorithm that makes polynomial calls to the subroutine. Then the runtime is still polynomial since polynomials are closed under composition. But suppose "efficient" were instead defined as quadratic time. When composing such an algorithm with another $O(n^2)$ algorithm, the new algorithm won't necessarily be efficient; it could take $O(n^4)$ time. Thus, we need something that's convenient for theory.

What are the actual limits of what we can do in polynomial time? Does it encompass what you would use computers for on a daily basis (arithmetic, spell checking, etc.)? Fortunately these are all polynomial time. For these tasks, it requires some thought to find the best polynomial time algorithm, but it doesn't require much thought to find *a* polynomial time algorithm. Our definition encompasses non obvious algorithms as well. You may be familiar with some of these if you have taken an algorithms class.

4 Longest Increasing Subsequence

4.1 The Problem

Let $X(1), X(2), ..., X(n)$ be a list of integers, let's say from 1 to $2n$. The task is to find the longest subsequence of numbers, not necessarily contiguous, that is increasing, i.e. $X_{i_1} < X_{i_2} < \ldots < X_{i_k}$ where $1 \leq i_1 < i_2 < \ldots < i_k \leq n$. For example:

$$3, 8, 2, 4, 9, 1, 5, 7, 6$$

has several longest subsequences of length 4: (2,4,5,7), (2,4,5,6), (3,4,5,6), (3,4,5,7).

Solving this is manageable when we have 9 numbers, but what about when $n = 1000$? How do we program a computer to do this?

4.2 A Polynomial Time Algorithm

One could try all possibilities, but this approach is not efficient. Trying all possibilities of size k takes $\binom{n}{k}$ time, and $\sum \binom{n}{i} = 2^n$. We'd have an exponential algorithm.

Consider this approach instead:

Iterate through the elements from left to right and maintain an array of size n. For each element i, save the longest subsequence that has i as its last element. Since computing this subsequence for the kth element involves only looking at the longest subsequences for the previous $k-1$ elements, this approach takes $O(n^2)$ time.

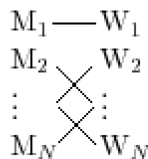
4.3 Dynamic Programming

The above was an example of *dynamic programming*, a general technique that allows us to solve some problems in polynomial time even though they seem to require exponential time. In general, you start with solutions to subproblems and gradually extend to a solution to the whole problem. However, for this technique to work, the problem needs to have some sort of structure that allows us to split it into subproblems.

5 Stable Marriage

5.1 The Problem

Given N men and N women where each man ranks each woman, and vice versa, we want to match people off in a way that will make everyone "not too unhappy."



More specifically, we want to avoid *instabilities*, where a man and woman who are not married both prefer each other over their actual spouses. A matching without any such instabilities is called a *stable marriage*. Our goal is to give an efficient algorithm to find stable marriages, but the first question is: does a stable marriage even always exist?

5.2 Victorian Romance Novel Algorithm

It turns out that the easiest way to show that a solution does exist is to give an algorithm that finds it. This algorithm, which was proposed by Gale and Shapley in the 1950's, has one of the simplest human interpretations among all algorithms.

5.2.1 Algorithm

Start with the men (this was the 50's, after all). The first man proposes to his top ranked woman and she tentatively accepts. The next man proposes, and so on, and each woman tentatively accepts. If there is never conflict, then we're done. If there is a conflict where 2 men propose to the same woman, the woman chooses man she likes better, and the man gets booted, crossing off the woman who rejected him. We keep looping through the men like this, where in the next round, any man who got booted goes to the next woman he prefers. If she hasn't been proposed to, she tentatively accepts. Otherwise, she picks which man she likes best, booting the other one, and so forth. We continue looping like this until all men and women are matched.

5.2.2 Termination

Does this algorithm terminate? Yes: in the worst case, every man would propose once to every woman on his list. (Note that a man never reconsiders a woman who's been crossed off.)

Next question: when the algorithm terminates is everyone matched up? Yes. Suppose for the sake of contradiction there's a couple who aren't matched. This means that there's a single man and a single woman. But in that case, no one else could've proposed to that woman, and hence when the man in question proposed to her (which he would have at some point) she would've accepted him. (Note that the problem statement encodes the assumption that everyone would rather be with *someone* than alone.) This is a contradiction; therefore everyone must be matched.

5.2.3 Correctness

Is the matching found by the algorithm a stable one? Yes. For suppose by contradiction that M_1 is matched to W_1 and M_2 to W_2, even though M_1 and W_2 both prefer each other over their spouses. In that case, M_1 would have proposed to W_2 *before* he proposed to W_1. And therefore, when the algorithm terminates W_2 couldn't possibly be matched with M_2—after all, she was proposed to by at least one man who she prefers over M_2. This is a contradiction; therefore the algorithm outputs a stable marriage.

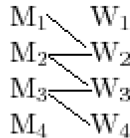
5.2.4 Runtime

Finally, note that the input consists of $2n$ lists, each with n numbers. The algorithm takes $O(n^2)$ time and is therefore linear-time with respect to the input size.

The naïve way to solve this problem would be to loop through all $n!$ possible matchings and output a stable one when found. The above is a much more efficient algorithm, and it *also* provides a proof that a solution exists.

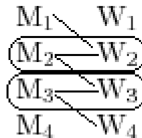
6 Other Examples

After forty years these are some of the problems we know are solvable in polynomial time: Given N men and N woman again, but no preference lists. Instead, we know whether M_i can be matched



to W_j, $1 \leq i, j \leq n$. It is clear that in this case we won't be able to make everyone happy. Take for example the case where no man can be matched with any woman. But even though we won't necessarily be able to make everyone happy, can we make as many people as happy as possible? This is called the "Maximum Matching" problem. More abstractly, given a bipartite graph with n vertices on each side, we want to find a maximal disjoint set of edges. The naïve way looks at how to match 1 person, 2 people, 3 people, etc. but this is slow.

A better solution involves repeatedly matching k people, and then trying to see if we "improve" the match to k+1 people. Examining the graph of men and women, if we find any men and women who are not matched then we tentatively match them off. Of course, following this approach, we might come to a situation where we can no longer match anyone off, and yet there might still exist



a better matching. We can then search for some path that zigzags back and forth between men and women, and that alternates between edges we haven't used and edges we have. If we find such a path, then we simply add all the odd-numbered edges to our matching and remove all the even-numbered edges. In this approach, we keep searching in our graph till no more such paths can be found. This approach leads to an $O(n^3)$ algorithm. As a function of the size of our input, which is n^2, this is an $O(n^{3/2})$ algorithm. Hopcroft and Karp later improved this to an $O(n^{5/4})$ algorithm.

It is worthwhile to note that this problem was solved in Edmonds' original 1965 paper that defined polynomial time as efficient. In that paper, Edmonds actually proved a much harder result: that even in the case where men could also be matched to men, and likewise for women, we can still find a maximal matching in polynomial time. Amusingly (in retrospect), Edmonds had to explain in a footnote why this was an interesting result at all. He pointed out that we need to distinguish between polynomial and exponential-time algorithms since exponential time algorithms don't convey as good an understanding of the problem and aren't efficient.

6.1 Gaussian Elimination

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Another example is solving linear system of equations. Instead of trying every possible vector, we can use Gaussian Elimination. This takes $O(n^2)$ time to zero out rows below, and doing this n

times till we get an upper triangular matrix takes a total of $O(n^3)$ time. Essentially, this method is a formalization of the idea of doing substitution.

What about first inverting the matrix A and then multiplying it by the vector y? The problem is that the standard way to invert a matrix is to use Gaussian Elimination! As an aside, it's been proven that whatever the complexity of inverting matrices is, it's equivalent to the complexity of matrix multiplication. A reduction exists between the two problems.

6.2 Linear Programming

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

What about solving a system of linear *inequalities*? This is a problem called linear programming, whose basic theory was developed by George Dantzig shortly after World War II. As an operations researcher during the war, Dantzig had faced problems that involved finding the best way to get shipments of various supplies to troops. The army had a large set of constraints on how much was available and where it could go. Once these constraints were encoded as linear inequalities, the problem was to determine whether or not a feasible solution existed. Dantzig invented a general approach for solving such problems called the simplex method. In this method, you start with a vector of candidate solutions and then increment it to a better solution with each step. The algorithm seemed to do extremely well in practice, but people couldn't prove much of anything about its running time. Then, in the 1970's, Klee and Minty discovered contrived examples where the simplex algorithm takes exponential time. On the other hand, in 1979 Khachiyan proved that a different algorithm (though less efficient than the simplex algorithm in practice) solves linear programming in polynomial time.

6.3 Aside

As a student in class pointed out, had it not been for World War II, computer science (like most other sciences) would probably not have made so many advances in such a short period of time. If you're interested in learning more about the history of wartime research (besides the Manhattan Project, which everyone knows about), check out *Alan Turing: The Enigma* by Andrew Hodges or *Endless Frontier* by G. Pascal Zachary. World War II might be described as the first "asymptotic war" (i.e., the first war on such a huge scale that mathematical theories became relevant to winning it).

6.4 Primality Testing

Primality testing is the following problem: given a number, is it prime or composite? The goal is to answer in time proportional to the number of digits in the number. This is crucial for generating RSA keys; and fortunately, polynomial-time algorithms for this problem do exist. We'll go into this in more detail later.

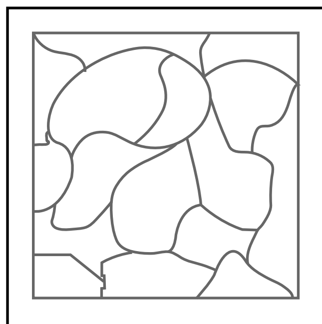


Figure by MIT OpenCourseWare.

6.5 Two-Coloring a Map

We're given a list of countries and told which countries are adjacent to which other ones. Can we color a map of the countries so that no two adjacent countries are colored the same way? A simple algorithm exists: color one country with one of the colors, and then color every adjacent country with the other color. We can keep going this way until either the map is fully colored or else two adjacent countries are forced to be the same color. In the latter case, the map is not two-colorable.

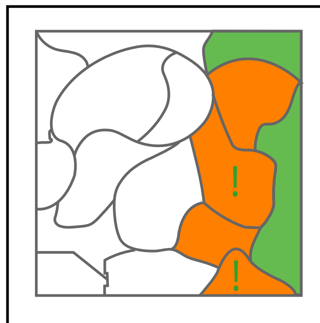


Figure by MIT OpenCourseWare.

What makes the problem so simple is once we color a country, we are forced in how we can color all the remaining countries. Because of this, we can decide whether the map has a two-coloring in $O(n)$ time.

6.6 Can we solve these in polynomial time?

On the other hand, what if we want to know whether our map has a *three*-coloring? This seems harder since we are no longer always forced to color each country a specific color—in many cases we'll have two choices. It's not obvious how to solve the problem efficiently.

Or consider a problem where we're given a list of people, and also told for any pair of people whether they're friends or not. Can we find a maximal set of people who are all friends with each other?

Stay tuned to next lecture to explore these problems...