Algorithms - Day 4

Instructor: Pat Devlin — prd41@math.rutgers.edu

Summer, 2016

A quick Fibonacci idea

On yesterday's homework, you found an algorithm for computing the n^{th} Fibonacci number, and it required a linear number of steps. Here's an *even faster* algorithm using matrices.

To multiply two 2×2 matrices, we use the following formula:

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{pmatrix}.$$

So for example, if M is the matrix $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$, then we can multiply M by itself a few times, and we see

$$M^2 = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}, \qquad M^3 = \begin{pmatrix} 1 & 2 \\ 2 & 3 \end{pmatrix}, \qquad M^4 = \begin{pmatrix} 2 & 3 \\ 3 & 5 \end{pmatrix}, \qquad M^5 = \begin{pmatrix} 3 & 5 \\ 5 & 8 \end{pmatrix}.$$

It turns out that for this particular matrix, we can find a simple formula for the entries of M^n involving the Fibonacci numbers! So if we want to find F_{100}, we could just figure out M^{100}, and then just use this formula to find our answer. This is then exactly like question 5 from yesterday's homework. In general, we can use this strategy to find F_n using only $\Theta(\lg(n))$ multiplications[1]!

Complexity theory

Now that we have the notion of 'running time,' we can meaningfully compare algorithms that address different problems. And more interestingly, we can even compare *problems* themselves.

Question 1 Consider the following problems.

(i) Sort a list of n numbers.

(ii) List all the subsets of $[n]$.

(iii) Find the maximum element of an unsorted list.

Which problems do you think are fundamentally easier for a computer to do, and which are harder? Explain.

[1] Although we usually think of multiplying numbers as a quick and easy process, in this situation, we may need to actually think a bit more about that because the numbers involved quickly become very large.

We know we can use Θ notation to compare the running times of different *algorithms*. And we can also compare *problems*.

Suppose we have two problems: A and B. To decide which of these problems is algorithmically more difficult, we want to compare the running time of the fastest algorithm solving A with the running time of the fastest algorithm solving B.

Question 2 When we compare two problems, why do we want to compare the *fastest* algorithm of one with the *fastest* algorithm for the other?

Definition 3 We say a problem, A, can be solved in *polynomial* time if and only if there is an algorithm solving A whose running time is $\mathcal{O}(n^c)$ for some $c > 0$.

Question 4 Which of the problems in question 1 can be solved in polynomial time? What are some other problems [e.g., from the homeworks or previous days] that can be solved in polynomial time?

There are a few great things about the question "can problem A be solved in polynomial time?"

- It's very useful to know if a problem can be solved in polynomial time, because that lets us know if the problem is easy or not.
- If we only care about the question "is there a polynomial time algorithm," then we don't really have to worry about a lot of the annoying difficulties that can arise when analyzing running time (e.g., 'what exactly *is* a step in this setting?').

Question 5 Try to think of a problem that *definitely cannot* be solved in polynomial time. [Be careful! Don't confuse the statement "I don't know how to solve this problem in polynomial time" with the statement "this problem cannot be solved in polynomial time."]

Here are a few "obvious" reasons that a problem couldn't be solved in polynomial time.

- If part of the problem *requires* the algorithm to output something whose size isn't a polynomial, then the problem couldn't possibly be solved in polynomial time. For example:
  - list all the subsets of $[n]$
  - list all the functions with domain $[n]$ and codomain $[n]$
- Because of the information theory lower bound, if there are N possible outputs of the algorithm, then the algorithm could not possibly finish in fewer than $\lg(N)$ steps. So if N is extremely large, then the problem couldn't possibly be solved in polynomial time. For example:
  - I'm thinking of a number less than 2^{2^n}. Which one is it?
  - Sort this list of length $n^{n!}$

So if we want to think about algorithmically *interesting problems*, we should definitely rule out these silly possibilities.

Definition 6 A decision problem is a problem where the answer is either yes or no.

Example 7 The following problems are decision problems because each is asking a yes/no question.

- Is this list of length n sorted?
- Is it possible to complete this partially filled in Sudoku puzzle?
- Does this graph with n vertices have an independent set of size k?

However, the problems "sort this list of size n" and "solve this Sudoku puzzle" are *not* decision problems because the output is not simply 'yes' or 'no.'

Question 8 What are some more examples of decision problems? What are some more examples of problems that *are not* decision problems?

Remark: Many problems that are not decision problems can be broken into pieces, where each piece *is* a decision problem.

Example 9 We can break up the problem "what's the size of the largest independent set of this graph" into multiple decision problems.

There are three basic types of decision problems.

- **Some decision problems are easy** in that these can be *solved* in polynomial time.

  The collection of these problems is called P.

  These are the problems that we *know* are easy.

- **Some decision problems are easy to solve after somebody gives you a big hint**. These are the problems where if the answer is in fact 'yes,' then God could convince us of this fact by giving us some sort of hint that we could check in polynomial time.

  The collection of these problems is called NP.

  These are the problems that are not *obviously* difficult.

  These are the problems that we can solve quickly if we have a big hint.

- **Some decision problems are so difficult that hints wouldn't even help us.** These are problems where even if God told us exactly what to look at, we still couldn't solve it in polynomial time.

  These are the problems that are *obviously* difficult.

- A problem is in P if and only if it can be solved in polynomial time
- A problem is in NP if and only if God could convince us that the answer is 'yes' in polynomial time

Example 10 The problem "is this list sorted" is in P because it can be solved in polynomial time. The problem "is this graph Eulerian" is in NP, because ...

Example 11 The following problems are in NP.

- Can all of these Tetris pieces fit into this square?
- Is this graph Hamiltonian?
- Does this 'instant insanity' puzzle have a solution?
- Is there a proper coloring of this graph using k colors?
- Is it even theoretically possible to beat this level of Super Mario Brothers?

Proposition 12 *Every problem in P is also in NP. In other words, $P \subseteq NP$.*

Proof:

Question 13 Does $P = NP$, or is there a problem in NP that cannot be solved in polynomial time?

This is literally the $1,000,000 question!