P and NP Fundamentals

Richard Carini and Nikola Kapamadzin

January 14, 2015

Algorithms[*]

Algorithms are a set of simple instructions for completing a task (Bartlett 2014). In computer science we program computers to go through algorithms to complete tasks. In mathematics we use algorithms to discuss the steps to formulating a proof or solving a problem. Here is a simple example of an algorithm that, using multiplication, tells us how to exponentiate to a positive degree.

Algorithm for exponentiating x^z[†]

```
1. Initialize $n = 1,\ x = y$
2. Let $y = x \cdot y$
3. Let $n = n + 1$
4. If $n = z$ then print(y)
   Else return to step two
```

Let's run through a quick example to show this works. Find 5^3 using this algorithm. Here we start at $x = 5,\ y = 5,\ n = 1,$ and $z = 3$
Now that we know where all of our variables start, let's go to step 2.

$$y = x \cdot y \implies y = 5 \cdot 5 \implies y = 25$$

Now that we have a value for y we go to step 3.

$$n = n + 1 \implies n = 1 + 1 \implies n = 2$$

Now we ask does n=z? We know that n=2 but z=3 therefore we go to else, and back to step 2.

$$y = x \cdot y \implies y = 5 \cdot 25 \implies y = 125$$

Then in step 3,

$$n = n + 1 \implies n = 2 + 1 \implies n = 3$$

Now we again ask, does n=z? We know n=3 and z=3 so n=z!
Therefore we print y, which is equal to 125.

[*] Written by Nikola
[†] Written by Nikola

Runtime [‡]

Runtime is a way to tell how long it takes to complete an algorithm. Runtime calculates the amount of "easy" or elementary operations that each take a fixed time to complete. Many algorithms exist that create a runtime that is too long, so finding efficient algorithms can be important.

Let's look at the runtime for our exponentiation algorithm. Assuming that it takes the same amount of time to add and multiply two numbers, it would take $z-1$ iterations of loop from steps 2 to 4 to get an answer. This is because we stop when $n = z$, start when $n = 1$, and change by adding one each time. Now in each one of these iterations we add once and multiply once. Therefore our runtime should be $2(z-1) = 2z-2$ units of time (or however long it takes to add/multiply). Therefore when we found 5^3 using this method, our runtime was 4, we added twice, and multiplied twice.

P and NP[§]

Now that we know the basic concepts of algorithms and their runtime, we can get to the meat of the question: What is P, what is NP, and what's the big deal with them anyway? We begin by defining these two terms P and NP

Definition 1. *P, the abbreviation for "Polynomial time," is the set of 'yes/no' or 'true/false' problems that are solvable with a polynomial runtime (Bartlett 2014).*

Definition 2. *NP, the abbreviation for "Nondeterministic polynomial time," is the set of all problems that can be checked in polynomial time (which includes problems that can be solved in polynomial time - namely P - but also adds in the problems that require exponential or factorial runtime) (Bartlett 2014).*

The previous example of exponentiation had a run-time that was in polynomial time, so as the size of the problem increases the number of steps needed is increased with a polynomial degree. Therefore, by our definition, we can see that the exponentiation of a number is in P. However, if a problem takes a polynomial time to verify an answer, but needs a large amount of time to solve (formally, an exponential amount of time - 2^n, or factorial amount of time - $n!$), the problem is in NP. We will now give an example of a problem that is in NP (but we will leave the proof for a later time): prime integer factorization. Recall that any positive integer can be written as the product of prime numbers. Since this is an inherent property for all numbers, we can focus on a sub category of this called "semiprimes."

Definition 3. *A semiprime is any product of two primes, denoted by P2.*

The question that is NP is the following:

[‡]Written by Nikola
[§]Written by Richard

Conjecture 1. *Factoring any semiprime P2 into its prime components is in NP.*

We can give a "hand-wave" proof that this is indeed an NP problem by testing a few numbers. 15, for example, is the product of the primes 3 and 5. We know this pretty readily, but if we were to use a computer for it, we would need to check all of the values under 15 to see if they multiply together to make 15. A larger prime, such as the number 589, is more difficult to solve for both a computer and a human mind. Perhaps through a little effort and some guessing, we will be able to figure the primes are 19 and 31. However, once we get a prime number that is 100 digits long, like the number 1522605027922533360535618378132637429718068114961380688657 908494580122963258952897654000350692006139[¶], our brain starts to ache. Fortunately, the computer feels the same way. To go through each one of the primes that are less than this number, let alone testing every value's divisibility under this number, requires about four hours in order to solve. However, checking to make sure that the two primes for this semiprime actually result in this prime is relatively easy. Your pocket calculator may have a difficult time displaying the number on the screen, but Wolfram Alpha would easily be able to verify the factorization. Because of the problem's inherent difficulty in solving, but quick ability to check, we can place this problem in NP. In fact, the complexity that surrounds this problem for very large values of semiprimes, and the complexity that surrounds factoring in general, is the basis for most web security today.

Sources

Bartlett, Padraic. *Lecture 1: P and NP*. 2014.

RSA Honor Roll. 5 March 1999. *http : //www.ontko.com/pub/rayo/primes/hr_rsa.txt*

[¶] The actual primes that multiply to this number are 37975227936943673922808872755445627854 565536638199 and 40094690950920881030683735292761468389214899724061 (RSA Honor Roll)