P vs NP
& Computational Complexity

Miles Turpin

MATH 89S

Professor Hubert Bray

P vs NP is one of the seven Clay Millennium Problems. The Clay Millenniums have been identified by the Clay Mathematics Institute as seven of the most difficult mathematical problems of our time, and each carries a reward of 1 million dollars for proving or disproving. Proving whether P = NP or P ≠ NP is widely considered to be the most important problem in theoretical Computer Science.

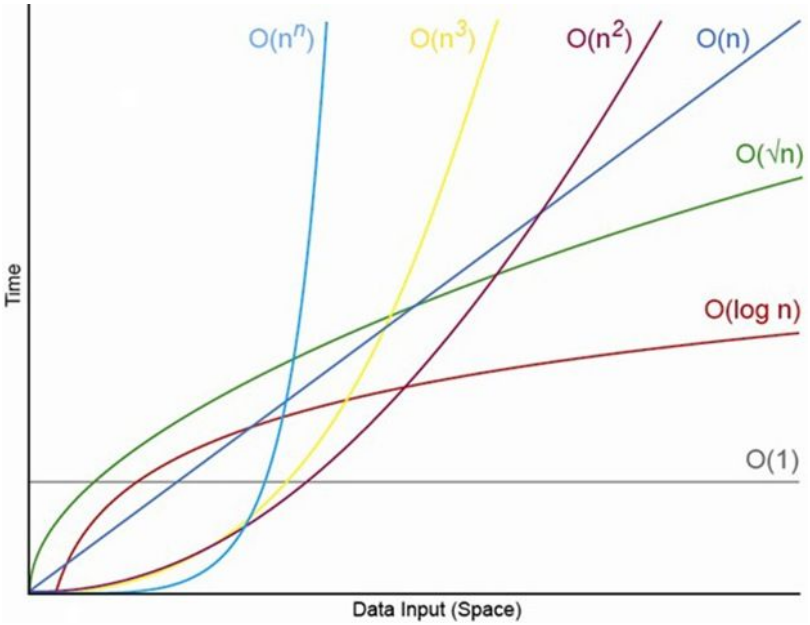
In short, P vs NP asks if all problems that have easy-to-check answers are also easy to solve. In discrete math jargon, P vs NP states whether problems whose solutions can be verified in polynomial time, have solutions that can be found in polynomial time. P (polynomial time) is the class of problems that have problems that can be found in polynomial time. NP (nondeterministic polynomial time) is the class of problems whose solutions can be verified in polynomial time. To properly understand the significance of this question, it will be helpful to consider discrete mathematics.

Discrete math is a branch of mathematics that deals with objects that assume distinct separate values, as opposed to objects with continuous values. An example of this distinction is the set of integers compared to the set of real numbers. Functions in calculus are smooth and differentiable while discrete math is concerned with computations on discrete states. Discrete mathematics is the language of computer science, as computers are “finite-state machines” (c.f. “infinite state machines” which have infinite memory and are theoretically possible but impractical). A state machine is any device that stores defined values or “states” at a given time and is capable of receiving inputs to manipulate those values.

The study of algorithms and how we solve problems is of particular importance to discrete math as it applies to computer science—computers have limits to their speed, and thus it is important that programmers design their programs to minimize the amount of computations required in a certain task. There are multiple ways to go about a computation, some being faster than others. Let’s take the multiplication of two numbers for example—there is a faster way to calculate the product of

two numbers than long multiplication. We use the term *complexity* to describe the relationship that describes how runtime (the time required to execute a task) correlates with the size of the inputs to the algorithm. We also talk about complexity in terms of order of magnitude, which is expressed in "Big O notation". This takes the form $O(f(N))$[1]. Because runtime can vary between hardware set ups, we maintain uniformity between machines by talking about time complexity in terms of the amount of number of computations executed in the algorithm.

With this context we are now in the position to understand polynomial time. Algorithms that execute in polynomial time have complexity $O(N^2)$, $O(N^3)$, $O(N^4)$, … , $O(N^m)$ for any number *m*. Algorithms that execute in polynomial time are generally considered to be acceptable speed, while those that execute in exponential time are incredibly slow. If an algorithm of complexity $O(N)$ takes 1 second to perform a calculation involving 100 elements, an algorithm of complexity $O(N^3)$ will take 3 hours, and an algorithm of complexity $O(2^N)$ will take *300 quintillion years* (Hardesty). This illustrates the reason we only talk about orders of magnitude--whether an algorithm is $O(0.5N^3)$ or even $O(N^4)$, these differences are nothing compared to an algorithm of $O(2^N)$.



Here are descriptions of programs that demonstrate each form of complexity:

[1] To be read: "on the order of some function of N"

- $O(1)$: a program receives a set of numbers that is size N and returns the value at index 27. Because the program can go straight to the 27th spot in the set without looking through the whole set, runtime does not vary with input size, thus $O(1)$.
- $O(N)$: a program searches through a set of numbers looking for the largest number in the set. This is $O(N)$ because the program must look through the entire set to determine which number is largest
- $O(N^2)$: suppose we wanted to find all the prime numbers from 1 to N. To test the primality for each number *n* (between 1 and N) we would divide it by every number from 2 to (*n* - 1), and if none of them give a whole number answer, then that number is prime—it has no factors. If we assume that division is an $O(1)$ operation for small enough numbers, then the complexity of dividing a number *n* by all those before it is an $O(n)$ operation. Now adding together these operations from 1 to N results in an overall complexity of $O(1) + O(2) + \ldots + O(N-1) + O(N)$ which with some simple math reduces to $O(N^2)$. I included sample Python code to run this program, as well as runtimes for the algorithm, for demonstration. The sentences in green are comments that might help explain the code.

```python
import time # this is used to calculate runtimes

maxNumberTestValues = [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000] # set up 10 values to test
primesDict = {}

for i in maxNumberTestValues:       # iterate through each test value, i will represent the test value in each iteration

    start = time.time()             # get time before prime-finding algorithm starts
    primes = []                     # create list to store the prime numbers in

    for j in range(1,i):    # iterate from 0 to maxNumber test value i where j is the number for that iteration

        isPrime = True              # start with assuming the value j is prime then only change to false if it has a factor

        for k in range (2, j - 1):  # iterate through every number from 2 to one less than the number we're checking for primality

            if (j % k) == 0:        # if k divides evenly into j (i.e. there is no remainder) the number j is not prime
                isPrime = False     # isPrime is false
            else:                   # if k does not divide evenly into j, keep going
                continue

        if isPrime == True:         # if isPrime is true after going through the loop, add the number j to the prime list
            primes.append(j)

    end = time.time()               # get time at end of algorithm
    runtime = end - start           # calculate runtime of algorithm

    print(str(i) + ': ' + str(runtime)) # print runtime for how long it takes to find primes for each test value i
    primesDict[i] = primes          # add the list of primes to a dictionary that associates the list of primes with test value i

for key, value in primesDict.items():
    print (key, value)              # print list of primes with associated test value

```

```
1000: 0.06581354141235352
2000: 0.2727932929992676
3000: 0.6264545917510986
4000: 1.1273956298828125
5000: 1.7776927947998047
6000: 2.5737361907958984
7000: 3.5151207447052
8000: 4.605397939682007
9000: 5.8673951625823975
10000: 7.257018327713013
```

This represents the runtimes of how long it took the program to find all primes from 1 to 1000, 1 to 2000 etc. Empirically we see the $O(N^2)$ relationship as when input size goes up by a factor of 10 from 1000 to 10000, runtime scales by a factor of roughly 100 from 0.0658 to 7.257 seconds.

```
1000 [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167,
173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269,
271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379,
383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487,
491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607,
613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727,
733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853,
857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977,
983, 991, 997]
```

- $O(2^N)$: a program prints all the possibilities of results of flipping a coin N times.

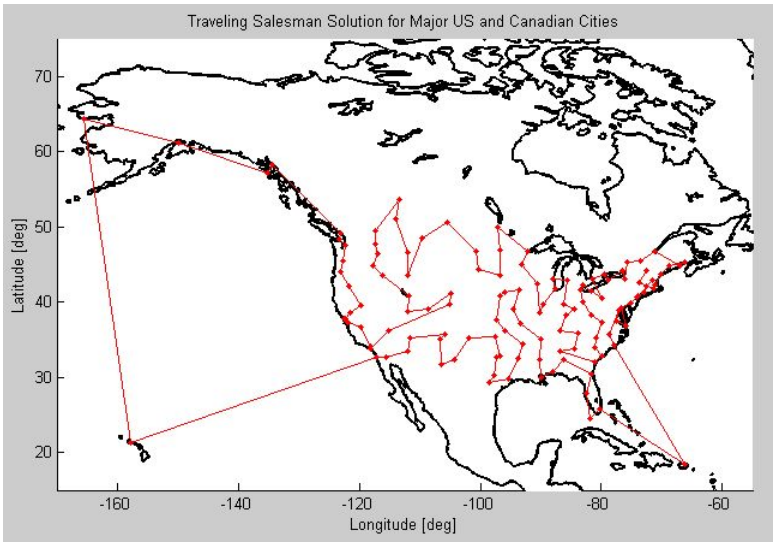
Returning to our multiplication example, long multiplication as we learn it in grade school is $O(N^2)$. Karatsuba multiplication was developed in 1960 and has complexity $O(N^{log_2 3}) = O(N^{1.585})$. As an illustration of what another multiplication algorithm might look like, consider Karatsuba multiplication as explained by Donald Knuth:

> "Suppose we want to multiply two 2-digit numbers: $x_1x_2 \cdot y_1y_2$:
>
> 1. compute $x_1 \cdot y_1$, call the result A
> 2. compute $x_2 \cdot y_2$, call the result B
> 3. compute $(x_1 + x_2) \cdot (y_1 + y_2)$, call the result C
> 4. compute $C - A - B$, call the result K; this number is equal to $x_1 \cdot y_2 + x_2 \cdot y_1$
> 5. compute $A \cdot 100 + K \cdot 10 + B$.
>
> Bigger numbers x_1x_2 can be split into two parts x_1 and x_2. Then the method works analogously. To compute these three products of m-digit numbers, we can employ the same trick again, effectively using recursion. Once the numbers are computed, we need to add them together (step 5.), which takes about n operations." (Knuth)

Today the fastest algorithm in use is the Schönhage–Strassen algorithm, which takes advantage of the Fourier transform to do multiplication. This has complexity $O(N\ log(N)\ log(\ log(N)))$ (Knuth).

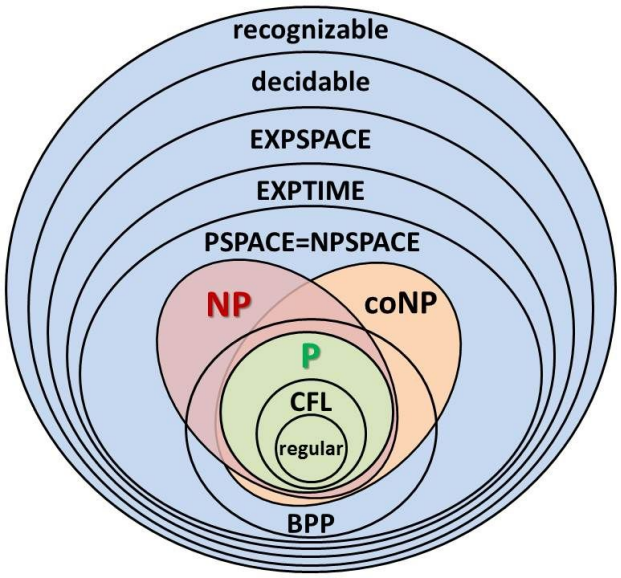
Returning to the question at hand, P vs NP states whether problems whose solutions can be verified in polynomial time, have solutions that can be found in polynomial time. The issue is that it would seem that most NP problems can only solved in exponential time. The most famous example of an NP problem is finding the prime factors of an incredibly large number—solving it requires checking an incredible number of possibilities while verifying requires just multiplying some numbers together. NP problems also show up in everyday life in the form of scheduling tasks. The travelling salesman problem is a notorious problem in computer science that asks to find the shortest route that passes through any set of points once and only once. Even coordinating which Uber comes to pick you up to

minimize driving time actually ends up being an incredibly complex scheduling nightmare. In fact, it's so difficult that the algorithms that Uber has developed to manage this are actually where the value in the company lies and why it's so hard to compete with them—no one can develop better solutions. These kinds of NP problems all belong to a subset of NP problems called NP-complete, which are also referred to as "hard search problems." This is because the only way to solve them is not through some mathematical trickery, but to simply try all the possibilities, which due to their setup, becomes incredibly time consuming.



There are many classes beyond P and NP that make up what we call the computational complexity zoo. In fact, many think that there is an infinite hierarchy of classes that we can make to distinguish different kinds of algorithms. Some of the most important ones are captured in this graphic below. P is contained within NP, that is because checking the solution to a P problem is as simple as doing out the problem, which is easy to begin with. The class NP-complete, which was brought up before, is a subset in NP, and any problem that is at least as hard as an NP complete problem is called NP Hard. coNP refers to the class of problems where it is easy to exclude wrong answers. Both NP and coNP are contained within PSPACE, problems which can theoretically be solved given an infinite amount of time but only given a polynomial amount of space. EXP problems take an exponential amount of time to both find and check the solution. An example of this is a question like

what is the best move to make in a game of chess? It takes an exponential amount of time to find the best move, but also to check it, since you only know if it is the best move by playing out the game. BPP is the class of problems that can be solved probabilistically in polynomial time. BQP is the analogue to this for quantum computing. Finally, some problems are impossible to solve given an infinite amount of time and space—that is the answer to the program is *undecidable*. The problem is recognizable but not decidable. The famous example of such a problem is the Halting Problem, which was proved to be impossible to solve in 1936 by Alan Turing. Turing proved that there is no universal program that when fed another program could tell whether that program would "halt" or go on forever.



It is tempting to speculate that quantum computers might help us show that P might in fact equal NP. Quantum computers take advantage of the property of quantum particles that allows them to be in multiple states at once, meaning that quantum computers would be able to perform multiple simultaneous calculations, resulting in a *massive* speedup in computing. The idea would be that with a quantum computer we could try all possibilities at once to solve an NP problem. Scott Aaronson, who specializes in quantum computational complexity at MIT (he's hardly qualified to answer this question), says this is not possible. NP≠BQP, which means that hard search problems can't be solved by a quantum computer in polynomial time (Aaronson). Quantum computers would only give us a polynomial speedup for NP problems. Even if they allowed us to make 100 times as many calculations as we can now in one shot, that might reduce our runtime in the problem mentioned at the beginning

of this paper from 300 quintillion years to 3 quintillion years—hardly helpful. There's is an upside to this however—cryptography would still be safe in quantum computer world. The RSA cryptography schema depends on the exact properties of traditionally NP-complete problems—problems that are hard to solve but easy to check the answer to. One computer will encrypt its information into a massive number and the only way to decrypt it is to know the prime factors of the number. If two computers know the prime factors of the number they can easily encrypt and decrypt, but a hacker intercepting the encrypted data will be unsuccessful at finding these prime factors by brute force.

Part of what makes P vs NP so compelling is that if a polynomial time solution can be found for one NP-complete problem, then it can be adapted to solve all the others. The implications of P indeed being equal to NP would mean our whole picture of the universe itself would change. Because of this, the general consensus is that $P \neq NP$. As Scott Aaronson puts it:

> "If P=NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett. It's possible to put the point in Darwinian terms: if this is the sort of universe we inhabited, why wouldn't we already have evolved to take advantage of it?"

Finding an efficient algorithm for NP-complete problems would mean miracle answers for protein folding that would help cure cancer and solutions for efficient markets that would revolutionize economics. So even if the general consensus is that P does not equal NP, then why has it been so hard to prove their inequality? Proving things is actually an NP problem. Although it is unlikely that the answer to this problem will revolutionize the world, finding a proof would deepen our understanding of computational complexity, as well as make the finder of the proof a million dollars richer.

Works Referenced

Aaronson, Scott. "Shtetl-Optimized." ShtetlOptimized RSS. N.p., n.d. Web. 29 Oct. 2016.

Hardesty, Larry. "Explained: P vs. NP." MIT News. Massachusetts Institute of Technology, 29 Oct. 2009. Web. 29 Oct. 2016.

D. Knuth, The Art of Computer Programming, vol. 2, sec. 4.3.3 (1998)