P $\stackrel{?}{=}$ NP, the problem to solve all problems

John Zhen

P $\stackrel{?}{=}$ NP is one of the most central questions in theoretical computer science, existing as a cornerstone of not only algorithmics but technological and societal advancement as a whole. Simply, it asks whether every problem with rapidly verifiable solutions can also be rapidly solved. However, beyond this seemingly straightforward mathematical puzzle is an enigma that has never yet been solved. It was recognised by the Clay Mathematics Institute as one of the seven greatest open problems of the millennium with a million dollar prize for the first proof, a feat which would hold profound importance to every field spanning from cryptography easily verifiable but impossible to reverse, to navigation systems estimating within seconds the shortest path out of thousands of roads and destinations, to even the foundations of mathematical reasoning itself.

1. Computational Complexity

Before defining the problem further, it is necessary to briefly explore computational complexity, a study which lies at the heart of P $\stackrel{?}{=}$ NP: given a problem, how should its "hardness" be quantified? One obvious approach would be to measure the time taken for a particular algorithm to run, but with different hardware and input sizes this is not particularly meaningful. As computing power grows exponentially (most modern machines perform over 100 billion calculations per second), many algorithms considered difficult a few decades ago can now be solved with extreme ease, yet some problems remain unyieldingly impossible – prime encryption for example takes orders of magnitude more than a human lifespan to break. Instead of considering absolute times, complexity can be analysed asymptotically to determine the number of operations as input size grows to infinity. This can be represented with Big-O notation, written:

$$O\big(f(n)\big) \text{ as } n \to \infty$$

where f describes how fast runtime grows in proportion to input size n. Common complexities are summarised below and illustrated with examples in JavaScript.

$\boldsymbol{O(1)}$

The fastest algorithms have constant complexity and run on $O(1)$ time. For example, accessing any array at an index only requires one operation regardless of its size.

```javascript
function elementAtIndex(arr, index){
  return arr[index];
}
```

$\boldsymbol{O(n)}$

As a linear function of n, runtime scales proportionately with input size. A simple algorithm with $O(n)$ time is calculating the sum of n numbers which will require n operations in total.

```javascript
function sum(arr){
  let total = 0;
  arr.forEach(n => total += n);
  return total;
}
```

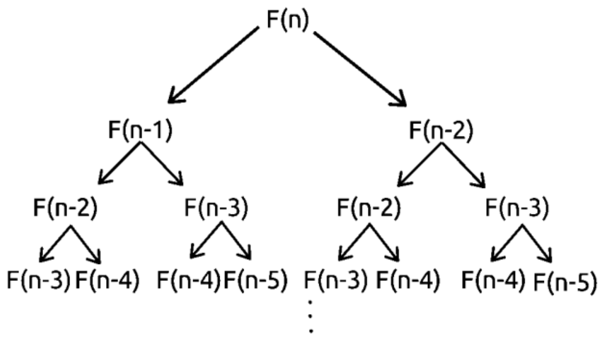
$\boldsymbol{O(n^k)}$

Polynomial time occurs when there are k nested loops present. For example, an algorithm to find a duplicate in an array contains two nested loops – if there are n elements in total, the inner loop will execute n times for each element and thus n^2 operations are performed overall, i.e. the algorithm runs on $O(n^2)$ time.

```
function findDuplicate(arr){
  for(i=0; i<arr.length; i++)
    for(j=i+1; j<arr.length; j++)
      if(arr[i] == arr[j]) return [i,j];
}
```

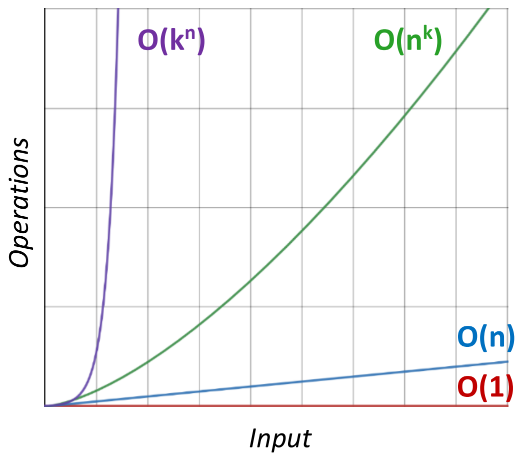
$O(k^n)$

Exponential time occurs when runtime increases with input size by a factor of k. With a recursive implementation, Fibonacci calculations grow exponentially – looking at the recurrence relation $F(n) = F(n-1) + F(n-2)$, its recursive tree can be constructed as shown on the right. As the number of nodes double at every level (2^0 on the first level, 2^1 on the second, 2^2 on the third, etc), an increase of n will double the additional operations required and thus the function has $O(2^n)$ time[1].



```
function fibonacci(n) {
    if(n <= 1) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

When time complexities are compared for general functions with large inputs (graphed on the right), it is clear that whilst the growth rate of polynomial time increases rapidly, they are dwarfed by that of exponential time algorithms which become vastly inefficient much sooner. This is true even for relatively small input sizes – at just $n = 100$, $O(n^2)$ requires 10^4 operations whilst $O(2^n)$ requires over 10^{30}. Thus, exponential time algorithms are generally to be avoided if a faster polynomial time approach exists, especially as in many practical applications n far exceeds 100.



2. Classifying P and NP

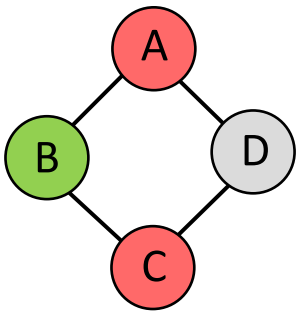
In P $\overset{?}{=}$ NP, P is defined as the set of all problems[2] solvable with a polynomial time algorithm (including constant and linear complexity), whereas NP is short for "non-deterministic polynomial time" and represents the set of all problems with solutions verifiable in polynomial time, no matter how long they take to solve. Note that if a problem exists in P, its solutions must also be verifiable in polynomial time, thus P is actually a subset of NP.

Take for example the 2-colourability problem – given a graph, determine whether every vertex can be assigned one of two colours (e.g. red and green) so that no two adjacent vertices have the same colour. This can be solved by colouring an arbitrary starting vertex red then traversing through the graph and

[1] This assumes the tree is symmetric, however in reality some branches are shorter than others and its time complexity will be lower than $O(2^n)$. The actual value for the base can be found with Binet's formula to be ϕ or the golden ratio (~1.618). Alternatively, just as the Fibonacci sequence has recurrence relation $F(n) = F(n-1) + F(n-2)$, so does its time complexity – the number of operations from one node equals the sum of its two branches below. This can be written as $k^n = k^{n-1} + k^{n-2}$ and dividing through by $k^n - 2$ then solving the resulting quadratic gives $k = \phi$.

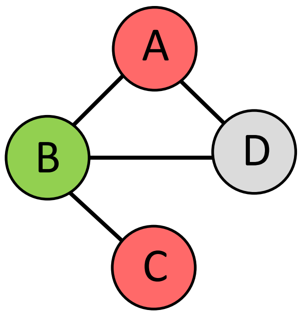
[2] Although formally P and NP only contain decision problems (problems which must yield a yes-or-no answer), it is possible to transform all other types of problems into decision problems and thus they will be generalised for the purpose of this essay..

alternating colours until either the entire graph is complete or the algorithm is forced to assign two adjacent vertices the same colour, in which case the graph is not 2-colourable.



2-colourable

D is adjacent only to red nodes and thus can be coloured green



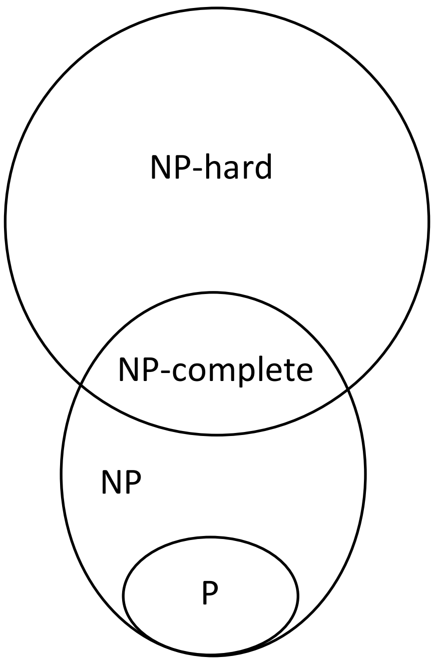
Not 2-colourable

D is adjacent to both red and green nodes and thus cannot be coloured into either

With V vertices and E edges, traversing through every vertex and edge only requires $O(V + E)$ linear time, hence 2-colourability is a member of P. Interestingly, extending to 3-colourability where vertices can be assigned to one of 3 colours, a polynomial time algorithm does not exist and instead graphs must be exhaustively searched with an inefficient backtracking approach, colouring vertices until no longer possible then changing the last colour and trying again. This has exponential time complexity of $O(3^n)$ for n total vertices and 3 possible colours assigned to each. However, any solution can be easily verified in $O(V + E)$ linear time by traversing through the graph and checking every vertex, thus 3-colourability is a member of NP but not P.

Conversely, many problems do not have solutions verifiable in polynomial time and are not members of NP, but instead a larger complexity class labelled NP-hard. An infamous optimisation problem existing in NP-hard is the Traveling Salesman Problem (TSP) which seeks to find the shortest path that travels through every vertex exactly once. Similar to 3-colourability, a backtracking approach with exponential time complexity must be used, however even when asked to verify a solution, it is impossible to confirm that a path is actually the shortest in polynomial time without redoing the problem.

NP-hard can be described as being intrinsically just as hard or harder than all problems in NP. Relative hardness can be determined using reduction – if it is possible to convert problem X into problem Y, then X is reducible to and must be no harder than Y. For example, 2-colourability can be reduced to 3-colourability, the harder version of the problem, simply by adding another available colour. Thus, a problem is NP-hard if it can be reduced from any problem in NP. In particular, the two sets overlap (as shown right) for some problems which have solutions verifiable in polynomial time whilst also having the key reduction property of NP-hard, i.e. they are the hardest to solve in NP. These problems are known as NP-complete and include 3 colourability, as well as more familiar problems such as Sudoku and Minesweeper which similarly can only be solved with backtracking by filling in squares until no longer possible. This is a common pattern in NP-complete – as their algorithms tend to be very closely related and often require some form of trial and error, if a polynomial time solution is found for one problem, it can be easily modified to solve other problems too.



3. Implications and importance of P $\stackrel{?}{=}$ NP

As any NP-complete problem can be converted into all of NP, they are crucial to answering P $\stackrel{?}{=}$ NP, however no polynomial time algorithm has yet been found despite decades of continual research. This supports the widely accepted conjecture that P ≠ NP – there exists some problems in NP that cannot be solved in polynomial time with any algorithm, although this remains unproven especially as existing proof techniques in complexity theory have been shown to be inapplicable to P $\stackrel{?}{=}$ NP and some researchers believe it is even independent of standard axiom systems. The power of NP-complete emerges when a single polynomial time algorithm is found, which through reduction can then be used to solve the entirety of NP. This will successfully prove that P = NP – every problem that can be verified in polynomial time can also somehow be solved in polynomial time. Whereas P ≠ NP (whilst being a significant academic feat) is already assumed to be true and would be inconsequential in practice, the impact of a constructive proof for P = NP would be immense due to the pervasiveness of computationally intractable problems across every industry. Key areas of computing rely on the asymmetry between problems difficult to solve but easy to verify, most notably public key encryption based on the products of large prime numbers impossible with current technology to factorise which along with other one-way functions would become vulnerable if a polynomial time algorithm were found. Whilst P = NP directly threatens cryptography, it provides solutions to optimisation problems which greatly benefit operational research in every field – graph traversal algorithms in navigation systems such as TSP which in polynomial time can currently only approximate the shortest path, register allocation (assigning variables in a program to physical memory) in CPUs based on k-colourability, and computing protein folds for minimum potential energy which is also NP-complete.

However, these breakthroughs pale in comparison to the revolution P = NP would bring to mathematics and human reasoning as a whole. As every mathematical discipline is foundationally built upon formal logic, the implications of polynomial time algorithms for intractable problems in algorithmics can be extended to that of mathematical theorems which are potentially difficult to prove but easy to verify for existing proofs (the key property of NP). Given any theorem T, this process can be transformed into a decision problem asking whether T is correct, the solutions of which constituting as a proof for T. Thus, P = NP suggests that proving a theorem would be no harder than verifying a valid proof. More significantly, it may be possible to generate proofs algorithmically simply by constructing a method to recognise them, although the form these solutions will take is not currently known, nor whether they will even be comprehendible by humans. Despite the primitiveness of automated deduction, many proofs have already been generated algorithmically, the first instance as early as 1976 with the 4-colour theorem which showed that every planar graph (with edges that cannot intersect) is 4-colourable by computing a set of every possible configuration. However, it was opposed by many researchers at the time and to this day remains controversial despite being a fully valid proof. Whilst human proof seeks elegance and rigour, the automated proof was calculated with brute force and lacks any real insight into the problem. Knowledge of a theorem's trueness alone is insufficient – human understanding is infinitely more valuable.

Similarly, if P = NP, it is believed that that creativity itself could potentially be automated if a method of verifying its results is found. A machine that recognises pieces by Chopin may theoretically be able to compose as Chopin did, or by identifying works by Van Gogh produce art to the same standard. AI is already an increasingly dominant contributor of creative content, generating images and texts indistinguishable from or even of higher quality than a human's. Although quality as an empirical concept cannot be emulated by mathematical processes and creativity is unlikely to ever be eradicated by P = NP, in a world where almost every problem can be solved algorithmically with extreme ease, inspiration and ingenuity becomes inefficient and ultimately archaic.

As problems demanded by civilisation increase both in number and complexity, $P = NP$ is utopia for knowledge and societal advancement. Instead of threatening understanding and creativity, by adopting a supporting role it has the potential to positively revolutionise every aspect of modern life. Despite the possibility that $P \overset{?}{=} NP$ will perhaps never be solved, it remains as the most important question in computer science of all time – any progress would bring significant insight into the abilities and limits of computation, and a proof of either $P = NP$ or $P \neq NP$ would be worth far greater than a million dollars.