INTERNATIONAL JOURNAL OF ADVANCE RESEARCH, IDEAS AND INNOVATIONS IN TECHNOLOGY

ISSN: 2454-132X
Impact factor: 6.078
(Volume 6, Issue 4)
Available online at: https://www.ijariit.com

P VERSUS NP PARADOX

Swostik Pati
swostikpati@gmail.com
Delhi Public School, Navi Mumbai, Maharashtra

ABSTRACT

The P vs. NP problem has haunted scientists' minds for years. It is considered to be one of the most difficult and deepest unanswered questions in the field of computer science and mathematics. Intellectuals all over the world have tried to reach a possible solution for years but there still isn't any consensus regarding the same. In this article, we will try to revisit the problem and discuss it using layman examples. I will touch all major aspects of computer science and mathematics related to the problem. Finally, I will conclude by providing a paradox that will cause people to think in a completely different way towards the possibility of a solution to the problem.

Keywords: *Polynomial Time, Computational complexity, Non-deterministic and deterministic algorithms, Conjecture, etc.*

1. THE MILLENNIUM PRIZE PROBLEMS

In the summer of the year 2000, the Clay Mathematics Institute gave the world the seven most profound and difficult problems in math that ever existed [1]. It announced a prize of 1 million dollars to be awarded to the discoverer(s) of the solutions to these problems. These problems were named as the Millennium Prize problems. The problems are listed as follows:

- Yang-Mills and Mass Gap
- Riemann Hypothesis
- P vs. NP Problem
- Navier-Stokes Equation
- Hodge Conjecture
- Poincare Conjecture
- Birch and Swinnerton-Dyer Conjecture

These problems are so difficult that it is beyond the scope of ordinary men to even understand the statement of some questions. The complexity of these problems can be inferred from the fact that to date, only one of these problems – the Poincare Conjecture - has been solved.

2. POLYNOMIAL AND EXPONENTIAL TIME

To understand this concept we will be taking real-life examples. Consider the situation when you come home from grocery shopping and are cross-checking the items you bought with the shopping bill. For this process, you take up an item in the bill and try finding it in the bag. For each item there are two steps: find it in the bill and then find it in the bag. This two-step process can be termed as iteration in the language of computer science. Each of these iterations has a loop nested in a loop (the two steps that we are following one after the other). So for example, we bought 10 items; this implies that there are 10 goods in the bag and 10 on the shopping list. For each iteration, we follow two steps, and hence the total steps involved will be of the order $10^2 = 100$.

Now, we visit another scenario where we try to guess a simple two-digit pin having numbers from 0-9 as inputs. It might feel quite similar to the previous case and that any person who can cross-check a grocery list can also crack this code. But in reality, it isn't possible. Let's breakdown this problem. In the beginning, we might see that there are two spaces and each takes 10 inputs. So the total number of permutations possible would be 10^2=100. But as we go down to computational levels we find that in order to guess one of these possible solutions, the algorithm goes through a lot more steps than just 100. While guessing, the program fixes one number in the first space and iterates through each of the 10 numbers in the second space. This process is repeated 10 times. Therefore, the total steps involved will be in the order of 10^{10}.

What made the two problems different?
On carefully comparing and generalizing the two situations we find this that when we increase the input size in the first case, the number of loops remains constant, and only the number of iteration increases. This, therefore, can be expressed in the form of 'n^k', where n is the input size and k is a constant quantity. But as we come down to the second problem, we find that on increasing the number of inputs, the number of nested loops grows as well. This can be expressed in the form of 'k^n', where n is the input size and k is a constant quantity.

In the world of computer science, time is measured, not in absolute quantities like seconds, but in relative units of the number of machine operations required to perform a task. The first type of problem is an example of Polynomial time (n^k) and the second type of problem is an example of Exponential time (k^n). The polynomial time problems are those which are considered to be solved within a reasonable amount of time. In contrast, exponential time problems are those which, even with

small sizes of inputs, are impossible for the fastest and most efficient computers to solve.

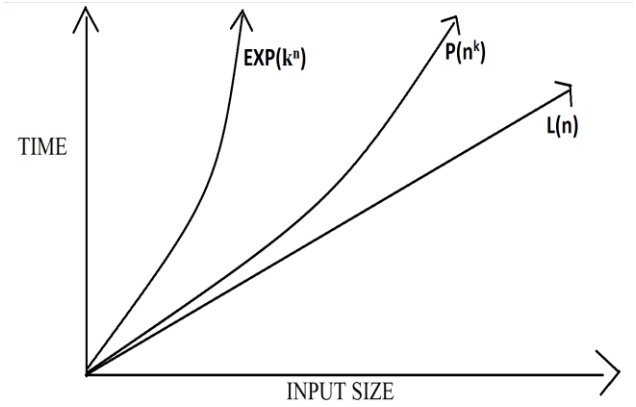

Fig. 1: Graphs of Exponential, Polynomial and Linear Time [20]

On plotting graphs for comparing linear time, polynomial time, and exponential time, we find that the slope of exponential time is much steeper than those of linear (n) and polynomial time [15].

3. P AND NP PROBLEMS

Before getting into the classes of complexity problems, we will first review the concept of decision problems.

A decision problem is one that returns either a 'yes' or 'no' as an answer. All problems can be reduced to decision problems. Even problems like sorting a group of numbers in ascending or descending order can also be expressed as a decision problem. Are the numbers sorted? There can be numerous decision problems made for the same problem. Solving the real problem, i.e. sorting, is much more difficult than the decision problems. But in order to solve the decision problems regarding sorting, we need to complete the sorting first. When a decision problem is solved, it is considered to be decided rather than solved.

Computer scientists have always tried to come out with algorithms to solve all the problems that exist in the world. Initially many problems had solutions that were almost impossible to be carried out even by a computer because of the time these algorithms needed. But soon enough, faster and more efficient programs were developed and the old ones were discarded. We can take the example of sorting again. Initially, algorithms like bubble sort (which were indeed fast) existed. Bubble sort had a time complexity of 'n^2'. But soon even a faster method of sorting, the merge sort was developed which had a time complexity of 'n log(n)'. There is always a search for algorithms that can sort with a time complexity of 'n'. But from this, we get to know that with time, faster and more efficient algorithms are discovered for the same problem.

With the increase in the number of algorithms, people soon realized that all of them could be categorized into specific categories: algorithms that were fast and could be solved in a reasonable amount of time and algorithms which were very slow and could never be solved in a reasonable time. There was also a class of problems which was considered completely unsolvable. We will be looking at each of them in detail.

3.1 Polynomial time problems – P

P was considered to be a class of all problems that could be solved in polynomial time. For example, addition and subtraction fall in the P category of problems. By formal definition, P is defined as the class of decision algorithms which can be 'decided' in polynomial time. These are all the decision problems that have algorithms fast enough to return a 'yes' or 'no' as an answer in a reasonable amount of time.

3.2 Non-Deterministic Polynomial time problems – NP

Consider a riddle. It may seem difficult to solve, but once the answer is given, it is easy to verify. NP is similar. NP is the class of decision problems that cannot be solved in polynomial time. But given a solution to these problems, there are algorithms that can verify the answer in polynomial time. For example, solving a 1000 x 1000 Sudoku grid. It might seem impossible, even for the most efficient computers. But once the solution is given, it can be easily verified within polynomial time.

3.3 Exponential time problems

Consider chess. How to determine which is the best move to make? Can the computers determine it?

We have seen computers beating chess grandmasters. This should be enough to make us believe that computers indeed can predict the next best move. Or is it?

Let's find out by first determining how many possible games are there in chess [14]. In the 1950s, Claude Shannon gave Shannon's number which estimated the number of games in chess. He considered that in each situation in the game, a player has about 30 legal moves (plys-a half move) on an average. Assuming a game to be of 40 moves he estimated that there are 10^{120} variations of the game of chess that are possible. This was a pretty rough estimate but even by these standards, if the fastest supercomputer had to determine which move was best, it would take millions of years to process each move.

Further down the line we now find that the number of games of chess was much more. In the beginning, white has 20 legal moves he can make. Then black will have 20 moves for each of the 20 variations white started with. That makes 400 moves in just one round. Taking it further, the numbers go like this: 8902 variations in the third move, 197,742 variations in the fourth, and so on. The largest chess game is considered to be somewhere around 11,800 plys long. By these standards, even the number of possible games cannot be predicted let alone the next best move. Also, given a move that is said to be the best move, we have no algorithm in place that can even verify whether this is true in polynomial time.

These are the type of problems which are considered to be unsolvable or take exponential time to be solved. Even if we have a possible solution to these problems, we lack algorithms that could verify the solution within polynomial time.

4. P VS NP

After the classification of problems in the specific categories, it was observed that many important problems that had great implications for humans laid in the class of NP. These were problems such as the Hamiltonian path problem, circuit designing (SAT), etc. which if solved in polynomial time, could be greatly beneficial for human society.

Scientists observed another trend as well. They were able to achieve faster algorithms for some of the problems in NP and reduce them to polynomial time. For example, the question of whether a number was prime or not, initially thought to be in

NP, was later discovered to be in P. But there remained other problems in NP for which no algorithms could be discovered to solve them in polynomial time.

Now, since all the problems that can be solved in polynomial time, can definitely be verified in polynomial time, P is considered to be a subset of NP. But whether or not the converse is true has left scholars in confusion for years.

Coming back to the millennium problem, the statement for P vs. NP goes as follows:
'*If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? '* (see [2])

This questions the fact that whether all the problems in NP can at some point be reduced to the class P, which further comes down to the question that-

Is P = NP ? or is P ≠ NP ?

Intellectuals all over the world have tried to devise algorithms that would make P equal to NP but there still exists no such algorithm. And judging from the fact that the number of problems in NP keeps increasing, most believe that P is not equal to NP. But even they have failed to prove it.

Now people might think why such a problem is even taken into consideration in fields which have many more important questions to deal with. The simple reason is if P was, in fact, equal to NP, then the way we look at the world will change completely. Every problem which could be verified in a reasonable amount of time could also be solved in a reasonable amount of time. The proofs in math which were based on observation could now actually be solved. The very basis of encryption and decryption will shatter. Passwords would be cracked in seconds. Problems like protein folding which required non polynomial time to be solved could easily be done in polynomial time, which would further go on to cure cancer. The possibilities are just endless. Of course, practically speaking all this won't be done by a computer algorithm. But we would then be having the capability of creating ways to achieve such feats.

Therefore the quest to find such an algorithm continues as scientists hope to solve the P vs. NP problem someday [3].

5. COMPUTATIONAL COMPLEXITY

Computational complexity refers to the process of classifying problems into complexity classes based on their inherent difficulty. Practically speaking, this allows us to understand the resource requirement of various algorithms [8] [9].

We previously discussed two classes:
P, which is the set of all problems solvable in polynomial time, and NP, which is the set of all problems verifiable in polynomial time.

There are many more sub-divisions and superclasses present in computational complexity. But before defining those, we will have to look at some important concepts.

5.1 Satisfiability

Boolean Satisfiability or SAT is one of the most studied problems in the whole of computer science. It is the problem for determining whether there exists an interpretation that satisfies a given Boolean formula.

CNF-Satisfiability is a simpler case of satisfiability where the Boolean formula has a specific form called 'Conjuncted Normal Form-CNF'.

Terminologies used:
- ^ - AND
- v - OR
- $\overline{\ }$ - NOT

The following is an example of a 3CNF-SAT problem [17]:

$$(XvYvZ) \wedge (Xv\overline{Y}vZ) \wedge (XvYv\overline{Z})$$

Here the expressions inside a bracket are joined at junctions. Each of these bracketed expressions is known as a clause which is joined with other clauses at conjunctions. The 3CNF-SAT problem consists of three conjuncted clauses.

The simple basis of the problem is determining for what values of X, Y, Z (if any) will the following algorithm return true. For a 3CNF-SAT problem, the time required to solve the problem is 2^3 and hence for an nCNF-SAT problem, the time required would be in the order of 2^n, which lies in the region of exponential time. Solving this algorithm is considered to be in NP as we can verify the algorithm if the solution is given to us.

5.2 Deterministic and Non-Deterministic Algorithms

A deterministic polynomial time algorithm is one that gives the same output for a given set of inputs. The use of each step of the algorithm can be determined.

In contrast, a non deterministic algorithm proceeds in a way that even the programmer himself can't determine. For the same set of inputs, it could provide different outputs in different runs.

When it was observed that the research to find algorithms to solve exponential time algorithms was going unfruitful, scientists tried to at least preserve their work for future research. So they wrote non-deterministic polynomial time algorithms of these exponential time problems. They knew how to proceed with a certain problem, but they couldn't figure out how to solve it in polynomial time. So they wrote deterministic algorithms with some lines of non-deterministic code where the program couldn't function in polynomial time. This way they could preserve their algorithms and when a polynomial time deterministic algorithm would be discovered, the non-deterministic parts from the code could be replaced.

This is what NP meant- Non Deterministic Polynomial Time algorithms [11].

5.3 NP-Hard

Another approach taken by scientists when no results from research were obtained was that of relating problems. This meant that instead of trying to solve all the problems in exponential time which proved to be unsuccessful, they started working on one single problem and relating all other problems to the base problem. The base problem was selected to be Satisfiability. This problem was called the NP-Hard problem.

The reason for relating all problems with the SAT was because this provided the scientists with a focused approach and also if they could solve Satisfiability, they could solve all other problems related to it. The method used for relating them was called reduction. Reduction meant that:

If satisfiability could be reduced to a problem Y, i.e. if the problem Y was some instance of the problem of satisfiability, and this reduction was possible in polynomial time, then the problem Y also belonged to the class of NP-Hard problems.

This property was also transitive. This meant that if satisfiability reduces to some problem Y and Y reduces to some problem Z and all the reductions take place in polynomial time, then Z also belongs to the class of NP-Hard problems. This way the list to NP-Hard started growing at a tremendous rate and soon enough all the major hard problems could be related using reduction.

5.4 NP-Complete

If an NP-Hard problem has a non-deterministic polynomial time algorithm, then the problem is said to be NP-Complete. Satisfiability has a proven non-deterministic polynomial time algorithm and hence is considered to be NP-Complete. This further infers that if a problem is reducible to satisfiability and has a non-deterministic polynomial time algorithm associated with it, then the problem also belongs to NP-Complete.

NP-Complete problems are considered to be the hardest problems in the class of NP. They are all related and hence solving even one of them would cause the class to instantly collapse and make P=NP.

In complexity theory, the Cook-Levin theorem states that any problem in NP can be reduced to polynomial time to the Boolean satisfiability problem [7]. This further goes on to say that if a deterministic algorithm can be discovered for Satisfiability, then every NP problem can be solved by a deterministic polynomial time algorithm.

So interestingly, the algorithm which could make solving Sudoku easier could also help in fast protein folding.

5.5 Major problems in NP-Complete

Some of the well known NP-Complete problems are given below [6]:

- Boolean Satisfiability Problem(SAT)
- Knapsack Problem – It is a type of optimization problem in which we are given certain weights having different known values. We need to put a certain number of these weights in a container such as to maximize the weight of the container, without exceeding the given limit.
- Hamiltonian Path Problem – A Hamiltonian path is one in which each vertex of a given set of vertices is touched one and only one time. The Hamiltonian path problem deals with determining whether or not a Hamiltonian path exists in a graph.
- Traveling Salesman Problem – It is another example of an optimization problem and also one of the most relatable ones for the general public. It basically questions that given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city? Solving this problem would make traveling for a salesman much easier.

The other important problems in NP-Complete are:

- Graph Coloring Problem
- Subset Sum Problem
- Subgraph Isomorphism Problem
- Independent Set Problem
- Clique Problem
- Vertex Cover Problem
- Dominating Set Problem

5.6 Final Overview of Computational Complexity

Over time, the complexity classes have grown to be much more complicated. The P vs. NP problem might be the one attracting the most attention, but there are even more vast complexity classes containing more difficult problems [10].

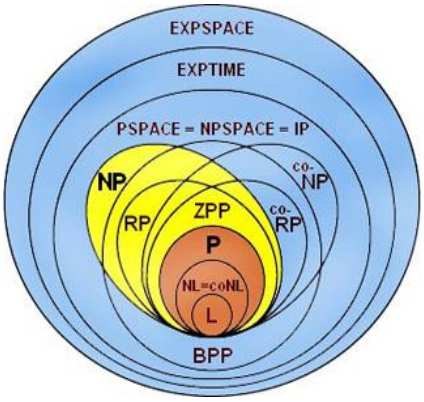

Fig. 2: Computational Complexity Classes [19]

The diagram shows many of these classes starting from P and extending far and wide. We will be defining each of them in brief:

- P is the class containing all the problems which can be solved by a deterministic algorithm in polynomial time.
- NP is the class containing all the problems which can be solved by a non-deterministic algorithm in Polynomial time.
- NP-Hard is the class containing all difficult problems which could be expressed as an instance of SAT.
- NP-Complete is the region of intersection between NP-Hard and NP.
- There is also another category of NP known as co-NP, which consists of decision problems where it is easier to prove one of the two possibilities as false and eliminate.
- Beyond NP, there are classes such as EXP, which represents exponential time.
- P Space represents the problems that require unlimited time to be solved but use only a polynomial amount of space.
- BPP (Bounded error Probabilistic Polynomial time) consists of problems that have polynomial time randomized algorithms as solutions. P is considered to be a subset of BPP [5].
- The quantum computing analog of BPP is BQP (Bounded error Quantum Polynomial time) [4].
- R is the class of all decision problems solvable in finite time.

The list just keeps growing. There are many smaller classes and groups present all over the complexity diagram. Many of the classes turn out to be infinite hierarchies of problems, with each one consisting of more problems than the ones within it. Many classes are thought to be the same and believed to collapse into each other. No one knows much except for the fact that there must be someplace where there would be a break between classes. There are also some problems which are yet to be solved like predicting the best move in chess, or are proven to be unsolvable like the famous Halting problem (the problem of determining, from a description of an arbitrary computer program and an input, whether the program will finish running, or continue to run forever [16].)

This list might be as endless as the problems themselves, or they might all collapse into one.

6. P VS NP AS AN NP PROBLEM

The P vs. NP problem is a proof in itself. It is a decision problem having two possibilities. Either P = NP or P ≠ NP.

Though most scientists believe that P ≠ NP for the sole reason that if P =NP was true, it would have been proved by now, due to the constant efforts of intellectuals. But this can be seen differently as well. Even after decades of research scientists still haven't been able to prove P ≠ NP [13].

So, just like any decision problem, both these possibilities have equal chances of occurring. It was further observed that it would be much easier to prove one of the two possibilities as wrong rather than proving any of them as right. So P vs. NP actually belonged to the class of Co-NP which is a subset of NP itself.

Therefore, the P vs. NP problem was indeed an NP problem in itself.

7. PARADOX

P vs. NP was always seen as a problem that would have a definite answer. But the point that the problem itself can be seen as a paradox is what I put forward as a hypothesis. From this point on, I shall be providing the paradoxical outlook of the P vs. NP problem.

Remember the P vs. NP problem can only be solved when either P = NP or P ≠ NP. Assume a situation in which a deterministic algorithm was found to solve all the problems in NP in polynomial time. That would maybe be termed as one of the greatest discoveries for human society. Scientists would be this close to solving the P vs. NP problem. Or would they be?

Even if all the problems in NP come down to P, there would still be one problem remaining in the class of NP, i.e. the P vs. NP problem itself. Since P won't be equal to NP unless both classes of P and NP collapse into one, this would imply that unless the P vs. NP problem is solved in polynomial time using a deterministic algorithm, P vs. NP problem won't be included in the class of P, and hence P=NP won't be possible. This means that out of the two possibilities in the P vs. NP problem, one is eliminated. The only other option remaining is

$$P \neq NP.$$

What did we just do? We just proved one of the two possibilities of the P vs. NP problem to be true by eliminating the only other option available. This means we just solved the P vs. NP problem.

Taking it even further, since P vs. NP was just solved in polynomial time, the P vs. NP problem would now be included in the class of P. This means that the only problem which remained out of the sphere of P and inside NP, got included inside P. So, we can now say that the classes of P and NP are the same ones.

We just proved P = NP. This means we solved the P vs. NP problem again.

Therefore, P is both equal to NP and not equal to NP at the same time. We again come back to the beginning having two possibilities of a decision problem that have an equal probability of being true. This means the P vs. NP problem is actually not solved.

In the language of superposition, it would mean that that the problem would both be solved and unsolved at the same time. This creates a paradoxical situation.

8. CONCLUSION

People might settle down for P ≠ NP after some years. Or maybe if we are fortunate enough, we would have made P = NP. But no matter what happens, the fact that the P vs. NP problem is a paradox in itself cannot be denied. The problem listed by the Clay Mathematics Institute actually has no definite solution but just paradoxical assumptions. It would remain as a Millennium problem forever. Though practically speaking, the quest to find a possible solution would benefit mankind greatly. But I believe that the P vs. NP problem would soon be termed as the P vs. NP PARADOX.

9. ACKNOWLEDGEMENT

I would like to thank my parents; Dr. Suvendu Kumar Pati and Mrs. Jyotirupa Kabi, for always believing in me and giving me hope to complete this research paper. I would also like to thank my younger brother, Ritwik Pati, for encouraging me and keeping me motivated to finish the paper. I also thank all my teachers who have made me capable enough to understand such a difficult problem and present to the world my views about the same. Lastly, I thank all my friends who made me believe in myself.

10. REFERENCES

[1] The Clay Mathematics Institute's web page for Millennium problems. https://www.claymath.org/millennium-problems
[2] The official statement for the P vs. NP problem. https://www.claymath.org/sites/default/files/pvsnp.pdf
[3] The Status of the P versus NP problem http://people.cs.uchicago.edu/~fortnow/papers/pnp-cacm.pdf
[4] BQP Wikipedia page https://en.wikipedia.org/wiki/BQP
[5] BPP Wikipedia page https://en.wikipedia.org/wiki/BPP_(complexity)
[6] NP- Completeness Wikipedia page https://en.wikipedia.org/wiki/NP-completeness
[7] Cook-Levin Theorem https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem#Contributions
[8] Computational Complexity Article by Stanford University https://plato.stanford.edu/entries/computational-complexity/
[9] MIT lecture on Complexity https://www.youtube.com/watch?v=mr1FMrwi6Ew&t=977s
[10] P vs. NP and the Computational Complexity Zoo By Hackerdashery https://www.youtube.com/watch?v=YX40hbAHx3s
[11] NP-Hard and NP-Complete ProblemsBy Abdul Bari https://www.youtube.com/watch?v=e2cF8a5aAhE
[12] P vs. NP - An Introduction By Undefined Behavior https://www.youtube.com/watch?v=OY41QYPI8cw&t=98s
[13] Donald Knuth: P=NP | AI Podcast Clips By Lex Fridman https://www.youtube.com/watch?v=XDTOs8MgQfg
[14] How many chess games are possible? By Numberphile https://www.youtube.com/watch?v=Km024eldY1A
[15] What is complexity theory? (P vs. NP explained visually) By Art of the Problem

https://www.youtube.com/watch?v=u2DLlNQiPB4
[16] The Turing & the Halting Problem By Computerphile https://www.youtube.com/watch?v=macM_MtS_w4
[17] JavaTpoint's web page for 3CNF SAT. https://www.javatpoint.com/daa-3-cnf-satisfiability
[18] Boolean Satisfiability Wikipedia page. https://en.wikipedia.org/wiki/Boolean_satisfiability_problem
[19] MITOpencourseware web page https://ocw.aprende.org/courses/mathematics/18-404j-theory-of-computation-fall-2006/
[20] What is complexity theory? (P vs. NP explained visually) By Art of the Problem – time stamp at 5:21 (Diagram) https://www.youtube.com/watch?v=u2DLlNQiPB4

BIOGRAPHY

Swostik Pati
High School Student
Delhi Public School, Navi Mumbai, Maharashtra

