Computational Complexity

## **Overview**

**“Computational complexity theory studies the complexity of algorithms and problems. It aims to identify the computational resources required to solve a problem, such as time, space, or communication, and to determine the limitations and possibilities of algorithmic efficiency.”**

One of the most important concepts in computational complexity theory is the P versus NP problem, which asks whether every problem that can be verified in polynomial time can also be solved in polynomial time. The class P consists of problems that can be solved in polynomial time, while the class NP consists of problems for which a solution can be verified in polynomial time.

NP-complete problems are a special subset of NP problems that are believed to be among the hardest problems in the class, in the sense that any NP-complete problem can be reduced to any other NP-complete problem in polynomial time. The notion of NP-completeness has important implications in cryptography, optimization, and many other areas of computer science and mathematics.

Other important concepts in computational complexity theory include the notion of polynomial-time reductions, which allows the complexity of one problem to be compared to another, and the complexity classes beyond P and NP, such as PSPACE and EXPTIME.

Overall, computational complexity theory is a rich and important field of study that provides a framework for understanding the computational limits and possibilities of algorithms and problems.

## **Solvable vs. Unsolvable Problems**

In computational complexity theory, problems are often classified as either solvable or unsolvable, based on whether there exists an algorithm that can solve the problem or not.

1. **Solvable Problem** - A problem is said to be solvable if there exists an algorithm that can solve it in a finite amount of time, i.e., for any input of the problem, the algorithm can produce the correct output in a finite number of steps. For example, the problem of finding the sum of two integers is solvable, since there exists a simple algorithm that can add two integers in a finite number of steps.
2. **Unsolvable Problem** - On the other hand, a problem is said to be unsolvable if there does not exist any algorithm that can solve it in a finite amount of time. One well-known example of an unsolvable problem is the Halting problem, which asks whether a given program will eventually halt or run forever. It has been proven that there is no algorithm that can solve the Halting problem for all possible inputs.

It is important to note that unsolvable problems are not the same as difficult problems. Some problems may be solvable, but require an exponential amount of time or space to solve, making them practically infeasible for large input sizes. The study of computational complexity theory focuses on understanding the limits of algorithmic efficiency, and identifying which problems are solvable, which problems are unsolvable, and which problems are solvable but infeasible.

## **Decidable vs. Undecidable Problems**

In computational complexity theory, solvable problems are also often classified as decidable or undecidable, based on whether there exists an algorithm that can decide the problem or not.

1. **Decidable Problem** - A problem is said to be decidable if there exists an algorithm that can determine whether an instance of the problem is a "yes" instance or a "no" instance in a finite amount of time. In other words, the algorithm will always halt and return the correct answer, either "yes" or "no". For example, the problem of determining whether a given number is prime is decidable, since there exists an algorithm that can determine the primality of a number in a finite amount of time.
2. **Undecidable Problem** - On the other hand, a problem is said to be undecidable if there is no algorithm that can determine whether an instance of the problem is a "yes" instance or a "no" instance in a finite amount of time. The classic example of an undecidable problem is the Halting problem, which asks whether a given program will eventually halt or run forever. It has been proven that there is no algorithm that can solve the Halting problem for all possible inputs.

It is worth noting that undecidability is a stronger notion than insolvability. All undecidable problems are unsolvable, but not all unsolvable problems are undecidable. For example, the problem of finding an optimal solution to the traveling salesman problem is unsolvable for large input sizes, but it is still decidable, since there exist algorithms that can find a suboptimal solution in a finite amount of time.

The study of decidable and undecidable problems is important for understanding the limits of algorithmic solvability, and for identifying the scope and limits of automated decision-making in computer science and other fields.

## **P vs. NP Problems**

In computational complexity theory, decidable problems are also often classified as P or NP problems.

The P versus NP problem is one of the most famous open problems in computer science and mathematics. It asks whether every problem that can be verified in polynomial time can also be solved in polynomial time. In other words, if a solution to a problem can be checked efficiently (in polynomial time), can it also be found efficiently (in polynomial time)?

The class of problems that can be solved in polynomial time is known as P, while the class of problems that can be verified in polynomial time is known as NP. Many important computational problems are known to be in NP, including the traveling salesman problem, the knapsack problem, and the satisfiability problem.

The P versus NP problem has important implications for cryptography, optimization, and other fields of computer science and mathematics. If P = NP, then it would be possible to solve many important computational problems much more efficiently than is currently known, and this could have major implications for fields such as cryptography and data analysis. However, if P ? NP, then many important computational problems may be fundamentally intractable, and this could have important implications for the limits of automated decision-making.

Despite decades of research, the P versus NP problem remains one of the most challenging open problems in computer science and mathematics, and it is not yet clear whether it will be resolved in the near future.

## **What are the major problems in Computational Complexity?**

There are several major problems in computational complexity theory that remain open and actively researched. Here are a few examples:

1. **P versus NP problem:** As I mentioned earlier, this is one of the most famous open problems in computer science. It asks whether every problem that can be verified in polynomial time can also be solved in polynomial time.
2. **Complexity of specific problems**: There are many important computational problems for which the complexity is not yet well understood. Examples include the traveling salesman problem, graph isomorphism, and factoring large integers.
3. **Circuit lower bounds:** Despite many efforts, there is still no known general technique for proving lower bounds on the size of Boolean circuits that compute specific functions. This is a major open problem in computational complexity theory.
4. **De-randomization:** Randomized algorithms are often more efficient than deterministic algorithms, but it is not known whether every randomized algorithm can be converted into an equally efficient deterministic algorithm.
5. **Quantum complexity theory:** Quantum computers have the potential to solve certain problems much faster than classical computers, but the study of quantum complexity theory is still in its early stages.

These are just a few examples of the many open problems in computational complexity theory. Despite their difficulty, progress in these areas can have significant implications for cryptography, optimization, machine learning, and other fields of computer science and mathematics.

Note: There are many important computational problems for which the complexity is not yet well understood. Examples include the traveling salesman problem, graph isomorphism, and factoring large integers.

# Advantage — The Problem-Solving Capabilities

The most common yet significant advantage of computational thinking is that it enhances problem-solving capabilities by leveraging the above-mentioned principles to aid learning. Using logic, computational thinking deducts new information/data based on current information, informing real-life conclusions rather than reaching assumptions.

With decomposition, you can break down complex problems into more manageable, easy-to-understand parts. Through pattern recognition, you seek similarities within or among difficulties. Additionally, there is an abstraction that solely focuses on the most pertinent details, ignoring any irrelevant information. And some algorithms develop step-by-step solutions to problems or provide rules that must be followed to create an appropriate solution.

Within educational settings, teachers and students can reinforce things like spelling rules through pattern recognition. Algorithms, meanwhile, can be used to create various writing styles, and abstraction can be used to boost research capabilities.

Computational thinking provides a reliable method to cope with different events, regardless of the industry, whether calculating numbers or growing fresh produce. It’s a multi-dimensional problem-solving concept.

# Advantage — Computational Thinking is Rooted in Research and Testing

The term computational thinking was coined by famed mathematician Seymour Papert and later emphasized by Jeannette Wing, bringing it to global attention with her research paper identifying the impact on society that computer science, algorithm design, and technology have. As a result of their philosophies, prominent world leaders and educational philosophers outlined computational thinking as a vital skill that opens peoples’ minds to data and resource usage. The skill shifts people from being technology consumers to becoming creators.

Given its deep roots in research and testing, established companies such as Oracle, Google, and Microsoft have long recruited staff to leverage computational thinking, providing them with a competitive edge.

# Advantage — Computational Thinking Promotes And Boosts Efficiency

Another key benefit that computational thinking offers is efficiency, minimizing the number of resources used for problem-solving purposes. Within computer science and algorithm design, resource minimization is vital to solving problems properly.

The time it takes for algorithms to solve problems and the memory space necessary to facilitate problem-solving are key to maximizing computational efficiency, with time being a great emphasis for many. Specific thought must go into algorithm designs to best handle specified problems rather than simply speeding up algorithm runtimes to best address time complexities. Efficient algorithms take less time and steps to solve problems, improving the productivity of the computational process.

With optimized algorithmic designs being logical aspects of the computational thinking process, computer instructions can be created using various languages that make machines and computers do things they were never previously capable of doing.

# Disadvantage — Difficulties with Prediction and Implementation

While computational thinking provides so many vast problem-solving opportunities for the people that use it, the predictability involved with computational thinking can sometimes be tricky.

With the computational thinking process, it may be difficult to accurately predict markets, trends, users, and all technical influences. As a result, there are too many variables involved that can complicate any given scenario and make it too difficult to model accurately.

Caching, where data is stored in cache memory, is one way to speed up the computational thinking process and make it easier. However, caching can be hard to integrate and necessitates the collection of the most accurate data for whatever the next instruction is.

Also, there are potential problems with the decomposition model in that an event-driven approach may not be possible compared to a procedural approach for programming purposes.

# Disadvantage — Knowing How Much Computational Thinking Aids Problem-Solving and Creativity

While applying computational thinking can be helpful in many settings, particularly in educational settings, there isn’t sufficient research that quantifies how much computational thinking helps. As a result, there’s no unequivocal measure of the range of its problem-solving abilities or how much it enhances creativity. Skills don’t automatically transfer, and computational thinking doesn’t definitively make someone better unless something is explicitly being taught to someone or a group of people.

In the end, I think, as more people and companies explore the capabilities and potential limitations of computational thinking, it’s clear that such a concept helps people develop sharper thought processes and connect with computers to solve problems effectively.