Computational complexity theory is a branch of the theory of computation in theoretical computer science that focuses on classifying computational problems according to their inherent difficulty, and relating those classes to each other. A computational problem is understood to be a task that is in principle amenable to being solved by a computer, which is equivalent to stating that the problem may be solved by mechanical application of mathematical steps, such as an algorithm.

A problem is regarded as inherently difficult if its solution requires significant resources, whatever the algorithm used. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication (used in communication complexity), the number of gates in a circuit (used in circuit complexity) and the number of processors (used in parallel computing). One of the roles of computational complexity theory is to determine the practical limits on what computers can and cannot do.

Closely related fields in theoretical computer science are analysis of algorithms and computability theory. A key distinction between analysis of algorithms and computational complexity theory is that the former is devoted to analyzing the amount of resources needed by a particular algorithm to solve a problem, whereas the latter asks a more general question about all possible algorithms that could be used to solve the same problem. More precisely, it tries to classify problems that can or cannot be solved with appropriately restricted resources. In turn, imposing restrictions on the available resources is what distinguishes computational complexity from computability theory: the latter theory asks what kind of problems can, in principle, be solved algorithmically.

A computational problem can be viewed as an infinite collection of instances together with a solution for every instance. The input string for a computational problem is referred to as a problem instance, and should not be confused with the problem itself. In computational complexity theory, a problem refers to the abstract question to be solved. In contrast, an instance of this problem is a rather concrete utterance, which can serve as the input for a decision problem. For example, consider the problem of primality testing. The instance is a number (e.g. 15) and the solution is "yes" if the number is prime and "no" otherwise (in this case "no"). Stated another way, the instance is a particular input to the problem, and the solution is the output corresponding to the given input.

To further highlight the difference between a problem and an instance, consider the following instance of the decision version of the traveling salesman problem: Is there a route of at most 2000 kilometres passing through all of Germany's 15 largest cities? The quantitative answer to this particular problem instance is of little use for solving other instances of the problem, such as asking for a round trip through all sites in Milan whose total length is at most 10 km. For this reason, complexity theory addresses computational problems and not particular problem instances.

When considering computational problems, a problem instance is a string over an alphabet. Usually, the alphabet is taken to be the binary alphabet (i.e., the set {0,1}), and thus the strings are bitstrings. As in a real-world computer, mathematical objects other than bitstrings must be suitably encoded. For example, integers can be represented in binary notation, and graphs can be encoded directly via their adjacency matrices, or by encoding their adjacency lists in binary.

Decision problems are one of the central objects of study in computational complexity theory. A decision problem is a special type of computational problem whose answer is either yes or no, or alternately either 1 or 0. A decision problem can be viewed as a formal language, where the members of the language are instances whose output is yes, and the non-members are those instances whose output is no. The objective is to decide, with the aid of an algorithm, whether a given input string is a member of the formal language under consideration. If the algorithm deciding this problem returns the answer yes, the algorithm is said to accept the input string, otherwise it is said to reject the input.

An example of a decision problem is the following. The input is an arbitrary graph. The problem consists in deciding whether the given graph is connected, or not. The formal language associated with this decision problem is then the set of all connected graphs—of course, to obtain a precisedefinition of this language, one has to decide how graphs are encoded as binary strings.

A function problem is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just yes or no. Notable examples include the traveling salesman problem and the integer factorization problem.

It is tempting to think that the notion of function problems is much richer than the notion of decision problems. However, this is not really the case, since function problems can be recast as decision problems. For example, the multiplication of two integers can be expressed as the set of triples (a, b, c) such that the relation a b = c holds. Deciding whether a given triple is a member of this set corresponds to solving the problem of multiplying two numbers.

To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem. However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve. Thus the time required to solve a problem (or the space required, or any measure of complexity) is calculated as a function of the size of the instance. This is usually taken to be the size of the input in bits. Complexity theory is interested in how algorithms scale with an increase in the input size. For instance, in the problem of finding whether a graph is connected, how much more time does it take to solve a problem for a graph with 2n vertices compared to the time taken for a graph with n vertices?

If the input size is n, the time taken can be expressed as a function of n. Since the time taken on different inputs of the same size can be different, the worst-case time complexity T(n) is defined to be the maximum time taken over all inputs of size n. If T(n) is a polynomial in n, then the algorithm is said to be a polynomial time algorithm. Cobham's thesis says that a problem can be solved with a feasible amount of resources if it admits a polynomial time algorithm.

A Turing machine is a mathematical model of a general computing machine. It is a theoretical device that manipulates symbols contained on a strip of tape. Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine—anything from an advanced supercomputer to a mathematician with a pencil and paper. It is believed that if a problem can be solved by an algorithm, there exists a Turing machine that solves the problem. Indeed, this is the statement of the Church–Turing thesis. Furthermore, it is known that everything that can be computed on other models of computation known to us today, such as a RAM machine, Conway's Game of Life, cellular automata or any programming language can be computed on a Turing machine. Since Turing machines are easy to analyze mathematically, and are believed to be as powerful as any other model of computation, the Turing machine is the most commonly used model in complexity theory.

A deterministic Turing machine is the most basic Turing machine, which uses a fixed set of rules to determine its future actions. A probabilistic Turing machine is a deterministic Turing machine with an extra supply of random bits. The ability to make probabilistic decisions often helps algorithms solve problems more efficiently. Algorithms that use random bits are called randomized algorithms. A non-deterministic Turing machine is a deterministic Turing machine with an added feature of non-determinism, which allows a Turing machine to have multiple possible future actions from a given state. One way to view non-determinism is that the Turing machine branches into many possible computational paths at each step, and if it solves the problem in any of these branches, it is said to have solved the problem. Clearly, this model is not meant to be a physically realizable model, it is just a theoretically interesting abstract machine that gives rise to particularly interesting complexity classes. For examples, see non-deterministic algorithm.

Many types of Turing machines are used to define complexity classes, such as deterministic Turing machines, probabilistic Turing machines, non-deterministic Turing machines, quantum Turing machines, symmetric Turing machines and alternating Turing machines. They are all equally powerful in principle, but when resources (such as time or space) are bounded, some of these may be more powerful than others.

Many machine models different from the standard multi-tape Turing machines have been proposed in the literature, for example random access machines. Perhaps surprisingly, each of these models can be converted to another without providing any extra computational power. The time and memory consumption of these alternate models may vary. What all these models have in common is that the machines operate deterministically.

However, some computational problems are easier to analyze in terms of more unusual resources. For example, a non-deterministic Turing machine is a computational model that is allowed to branch out to check many different possibilities at once. The non-deterministic Turing machine has very little to do with how we physically want to compute algorithms, but its branching exactly captures many of the mathematical models we want to analyze, so that non-deterministic time is a very important resource in analyzing computational problems.

For a precise definition of what it means to solve a problem using a given amount of time and space, a computational model such as the deterministic Turing machine is used. The time required by a deterministic Turing machine M on input x is the total number of state transitions, or steps, the machine makes before it halts and outputs the answer ("yes" or "no"). A Turing machine M is said to operate within time f(n), if the time required by M on each input of length n is at most f(n). A decision problem A can be solved in time f(n) if there exists a Turing machine operating in time f(n) that solves the problem. Since complexity theory is interested in classifying problems based on their difficulty, one defines sets of problems based on some criteria. For instance, the set of problems solvable within time f(n) on a deterministic Turing machine is then denoted by DTIME(f(n)).

Analogous definitions can be made for space requirements. Although time and space are the most well-known complexity resources, any complexity measure can be viewed as a computational resource. Complexity measures are very generally defined by the Blum complexity axioms. Other complexity measures used in complexity theory include communication complexity, circuit complexity, and decision tree complexity.

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size n may be faster to solve than others, we define the following complexities:

For example, consider the deterministic sorting algorithm quicksort. This solves the problem of sorting a list of integers that is given as the input. The worst-case is when the input is sorted or sorted in reverse order, and the algorithm takes time O(n2) for this case. If we assume that all possible permutations of the input list are equally likely, the average time taken for sorting is O(n log n). The best case occurs when each pivoting divides the list in half, also needing O(n log n) time.

To classify the computation time (or similar resources, such as space consumption), one is interested in proving upper and lower bounds on the minimum amount of time required by the most efficient algorithm solving a given problem. The complexity of an algorithm is usually taken to be its worst-case complexity, unless specified otherwise. Analyzing a particular algorithm falls under the field of analysis of algorithms. To show an upper bound T(n) on the time complexity of a problem, one needs to show only that there is a particular algorithm with running time at most T(n). However, proving lower bounds is much more difficult, since lower bounds make a statement about all possible algorithms that solve a given problem. The phrase "all possible algorithms" includes not just the algorithms known today, but any algorithm that might be discovered in the future. To show a lower bound of T(n) for a problem requires showing that no algorithm can have time complexity lower than T(n).

Upper and lower bounds are usually stated using the big O notation, which hides constant factors and smaller terms. This makes the bounds independent of the specific details of the computational model used. For instance, if T(n) = 7n2 + 15n + 40, in big O notation one would write T(n) = O(n2).

Of course, some complexity classes have complicated definitions that do not fit into this framework. Thus, a typical complexity class has a definition like the following:

But bounding the computation time above by some concrete function f(n) often yields complexity classes that depend on the chosen machine model. For instance, the language {xx | x is any binary string} can be solved in linear time on a multi-tape Turing machine, but necessarily requires quadratic time in the model of single-tape Turing machines. If we allow polynomial variations in running time, Cobham-Edmonds thesis states that "the time complexities in any two reasonable and general models of computation are polynomially related" (Goldreich 2008, Chapter 1.2). This forms the basis for the complexity class P, which is the set of decision problems solvable by a deterministic Turing machine within polynomial time. The corresponding set of function problems is FP.

Many important complexity classes can be defined by bounding the time or space used by the algorithm. Some important complexity classes of decision problems defined in this manner are the following:

Other important complexity classes include BPP, ZPP and RP, which are defined using probabilistic Turing machines; AC and NC, which are defined using Boolean circuits; and BQP and QMA, which are defined using quantum Turing machines. #P is an important complexity class of counting problems (not decision problems). Classes like IP and AM are defined using Interactive proof systems. ALL is the class of all decision problems.

For the complexity classes defined in this way, it is desirable to prove that relaxing the requirements on (say) computation time indeed defines a bigger set of problems. In particular, although DTIME(n) is contained in DTIME(n2), it would be interesting to know if the inclusion is strict. For time and space requirements, the answer to such questions is given by the time and space hierarchy theorems respectively. They are called hierarchy theorems because they induce a proper hierarchy on the classes defined by constraining the respective resources. Thus there are pairs of complexity classes such that one is properly included in the other. Having deduced such proper set inclusions, we can proceed to make quantitative statements about how much more additional time or space is needed in order to increase the number of problems that can be solved.

The time and space hierarchy theorems form the basis for most separation results of complexity classes. For instance, the time hierarchy theorem tells us that P is strictly contained in EXPTIME, and the space hierarchy theorem tells us that L is strictly contained in PSPACE.

Many complexity classes are defined using the concept of a reduction. A reduction is a transformation of one problem into another problem. It captures the informal notion of a problem being at least as difficult as another problem. For instance, if a problem X can be solved using an algorithm for Y, X is no more difficult than Y, and we say that X reduces to Y. There are many different types of reductions, based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions, and the bound on the complexity of reductions, such as polynomial-time reductions or log-space reductions.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

This motivates the concept of a problem being hard for a complexity class. A problem X is hard for a class of problems C if every problem in C can be reduced to X. Thus no problem in C is harder than X, since an algorithm for X allows us to solve any problem in C. Of course, the notion of hard problems depends on the type of reduction being used. For complexity classes larger than P, polynomial-time reductions are commonly used. In particular, the set of problems that are hard for NP is the set of NP-hard problems.

If a problem X is in C and hard for C, then X is said to be complete for C. This means that X is the hardest problem in C. (Since many problems could be equally hard, one might say that X is one of the hardest problems in C.) Thus the class of NP-complete problems contains the most difficult problems in NP, in the sense that they are the ones most likely not to be in P. Because the problem P = NP is not solved, being able to reduce a known NP-complete problem, $\Pi2$, to another problem, $\Pi1$, would indicate that there is no known polynomial-time solution for $\Pi1$. This is because a polynomial-time solution to $\Pi1$ would yield a polynomial-time solution to $\Pi2$. Similarly, because all NP problems can be reduced to the set, finding an NP-complete problem that can be solved in polynomial time would mean that P = NP.

The complexity class P is often seen as a mathematical abstraction modeling those computational tasks that admit an efficient algorithm. This hypothesis is called the Cobham–Edmonds thesis. The complexity class NP, on the other hand, contains many problems that people would like to solve efficiently, but for which no efficient algorithm is known, such as the Boolean satisfiability problem, the Hamiltonian path problem and the vertex cover problem. Since deterministic Turing machines are special non-deterministic Turing machines, it is easily observed that each problem in P is also member of the class NP.

The question of whether P equals NP is one of the most important open questions in theoretical computer science because of the wide implications of a solution. If the answer is yes, many important problems can be shown to have more efficient solutions. These include various types of integer programming problems in operations research, many problems in logistics, protein structure prediction in biology, and the ability to find formal proofs of pure mathematics theorems. The P versus NP problem is one of the Millennium Prize Problems proposed by the Clay Mathematics Institute. There is a US$1,000,000 prize for resolving the problem.

It was shown by Ladner that if $P \neq NP$ then there exist problems in NP that are neither in P nor NP-complete. Such problems are called NP-intermediate problems. The graph isomorphism problem, the discrete logarithm problem and the integer factorization problem are examples of problems believed to be NP-intermediate. They are some of the very few NP problems not known to be in P or to be NP-complete.

The graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic. An important unsolved problem in complexity theory is whether the graph isomorphism problem is in P, NP-complete, or NP-intermediate. The answer is not known, but it is believed that the problem is at least not NP-complete. If graph isomorphism is NP-complete, the polynomial time hierarchy collapses to its second level. Since it is widely believed that the polynomial hierarchy does not collapse to any finite level, it is believed that graph isomorphism is not NP-complete. The best algorithm for this problem, due to Laszlo Babai and Eugene Luks has run time 2O(√(n log(n))) for graphs with n vertices.

The integer factorization problem is the computational problem of determining the prime factorization of a given integer. Phrased as a decision problem, it is the problem of deciding whether the input has a factor less than k. No efficient integer factorization algorithm is known, and this fact forms the basis of several modern cryptographic systems, such as the RSA algorithm. The integer factorization problem is in NP and in co-NP (and even in UP and co-UP). If the problem is NP-complete, the polynomial time hierarchy will collapse to its first level (i.e., NP will equal co-NP). The best known algorithm for integer factorization is the general number field sieve, which takes time O(e(64/9)1/3(n.log 2)1/3(log (n.log 2))2/3) to factor an n-bit integer. However, the best known quantum algorithm for this problem, Shor's algorithm, does run in polynomial time. Unfortunately, this fact doesn't say much about where the problem lies with respect to non-quantum complexity classes.

Many known complexity classes are suspected to be unequal, but this has not been proved. For instance $P \subseteq NP \subseteq PP \subseteq PSPACE$, but it is possible that P = PSPACE. If P is not equal to NP, then P is not equal to PSPACE either. Since there are many known complexity classes between P and PSPACE, such as RP, BPP, PP, BQP, MA, PH, etc., it is possible that all these complexity classes collapse to one class. Proving that any of these classes are unequal would be a major breakthrough incomplexity theory.

Along the same lines, co-NP is the class containing the complement problems (i.e. problems with the yes/no answers reversed) of NP problems. It is believed that NP is not equal to co-NP; however, it has not yet been proven. It has been shown that if these two complexity classes are not equal then P is not equal to NP.

Similarly, it is not known if L (the set of all problems that can be solved in logarithmic space) is strictly contained in P or equal to P. Again, there are many complexity classes between the two, such as NL and NC, and it is not known if they are distinct or equal classes.

Problems that can be solved in theory (e.g., given large but finite time), but which in practice take too long for their solutions to be useful, are known as intractable problems. In complexity theory, problems that lack polynomial-time solutions are considered to be intractable for more than the smallest inputs. In fact, the Cobham–Edmonds thesis states that only those problems that can be solved in polynomial time can be feasibly computed on some computational device. Problems that are known to be intractable in this sense include those that are EXPTIME-hard. If NP is not the same as P, then the NP-complete problems are also intractable in this sense. To see why exponential-time algorithms might be unusable in practice, consider a program that makes 2n operations before halting. For small n, say 100, and assuming for the sake of example that the computer does 1012 operations each second, the program would run for about 4 1010 years, which is the same order of magnitude as the age of the universe. Even with a much faster computer, the program would only be useful for very small instances and in that sense the intractability of a problem is somewhat independent of technological progress. Nevertheless, a polynomial time algorithm is not always practical. If its running time is, say, n15, it is unreasonable to consider it efficient and it is still useless except on small instances.

What intractability means in practice is open to debate. Saying that a problem is not in P does not imply that all large cases of the problem are hard or even that most of them are. For example, the decision problem in Presburger arithmetic has been shown not to be in P, yet algorithms have been written that solve the problem in reasonable times in most cases. Similarly, algorithms can solve the NP-complete knapsack problem over a wide range of sizes in less than quadratic time and SAT solvers routinely handle large instances of the NP-complete Boolean satisfiability problem.

Before the actual research explicitly devoted to the complexity of algorithmic problems started off, numerous foundations were laid out by various researchers. Most influential among these was the definition of Turing machines by Alan Turing in 1936, which turned out to be a very robust and flexible simplification of a computer.

As Fortnow & Homer (2003) point out, the beginning of systematic studies in computational complexity is attributed to the seminal paper "On the Computational Complexity of Algorithms" by Juris Hartmanis and Richard Stearns (1965), which laid out the definitions of time and space complexity and proved the hierarchy theorems. Also, in 1965 Edmonds defined a "good" algorithm as one with running time bounded by a polynomial of the input size.

Earlier papers studying problems solvable by Turing machines with specific bounded resources include John Myhill's definition of linear bounded automata (Myhill 1960), Raymond Smullyan's study of rudimentary sets (1961), as well as Hisao Yamada's paper on real-time computations (1962). Somewhat earlier, Boris Trakhtenbrot (1956), a pioneer in the field from the USSR, studied another specific complexity measure. As he remembers:

Even though some proofs of complexity-theoretic theorems regularly assume some concrete choice of input encoding, one tries to keep the discussion abstract enough to be independent of the choice of encoding. This can be achieved by ensuring that different representations can be transformed into each other efficiently.

In 1967, Manuel Blum developed an axiomatic complexity theory based on his axioms and proved an important result, the so-called, speed-up theorem. The field really began to flourish in 1971 when the US researcher Stephen Cook and, working independently, Leonid Levin in the USSR, proved that there exist practically relevant problems that are NP-complete. In 1972, Richard Karp took this idea a leap forward with his landmark paper, "Reducibility Among Combinatorial Problems", in which he showed that 21 diverse combinatorial and graph theoretical problems, each infamous for its computational intractability, are NP-complete.