Algorithms

P vs NP

Polynomial Time

Polynomial time is slow — When talking about time complexity (i.e. how much time an algorithm takes to run), the programming community measures the time an algorithm takes based on input size n. The mathematical notation that describes the limiting behaviour of a function when an argument tends towards a particular value or infinity is called Big O notation. Generally, any algorithm that runs in polynomial time (i.e. takes n^k time given input of size n) is considered slow but still ‘accepted’ by the community as the upper limit. Anything slower is deemed not usable by the programming community as time taken scales up too fast relative to input size.

P

P (polynomial time) refers to the class of problems that can be solved by an algorithm in polynomial time. Problems in the P class can range from anything as simple as multiplication to finding the largest number in a list. They are the relatively ‘easier’ set of problems.

NP

NP (non-deterministic polynomial time) refers to the class of problems that can be solved in polynomial time by a non-deterministic computer. This is essentially another way of saying “If I have unlimited compute power (i.e. as many computers as I need), I am capable of solving any problem in at most polynomial time”. More intuitively though, it refers to the class of problems that currently, has no way of finding a quick (polynomial time) enough answer, BUT can be quickly **verified** (in polynomial time) if one provides the solution to the problem. The term verified here means that one is able to check that the solution provided is indeed correct.

Example

One of the most common yet effective examples is *Sudoku*. Given an unsolved Sudoku grid (9 x 9 for example), it would take an algorithm a fair amount of time to solve one. However, if the 9 x 9 grid increases to a 100 x 100 or 10,000 x 10,000 grid, the time it would take to solve it would increase **exponentially** because the problem itself becomes significantly harder. However, given a solved Sudoku grid (of 9 x 9), it is fairly straightforward to verify that the particular solution is indeed correct even if the size scales to 10,000 by 10,000. It would be slower, but the time to check a solution increases at a slower rate (**polynomially**).

Example

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision-makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

Example

The travelling salesman problem (also called the travelling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

NP Hard Problem

A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.

NP Complete Problem

A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time. NP-Complete problems are as hard as NP problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic Turing machine can solve NP-Complete problem in polynomial time.