Lesson 2

Topic: Intractability II (P, NP and NP-Complete)

Introduction (10 mins)

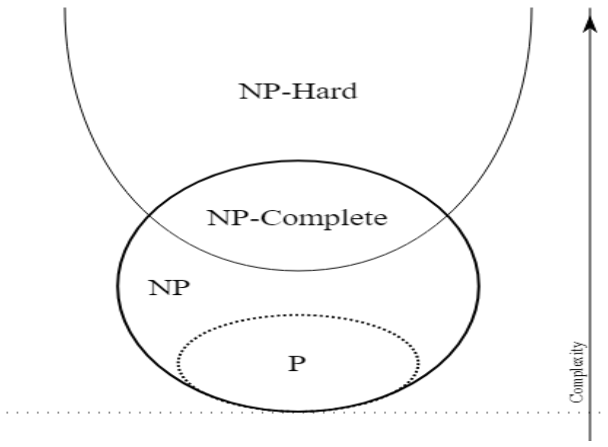
Computational Complexity Theory, often referred to simply as Complexity Theory, is a branch of theoretical computer science that studies the resources (such as time and space) required to solve computational problems. It deals with classifying problems based on their inherent difficulty and understanding the relationships between different classes of problems.

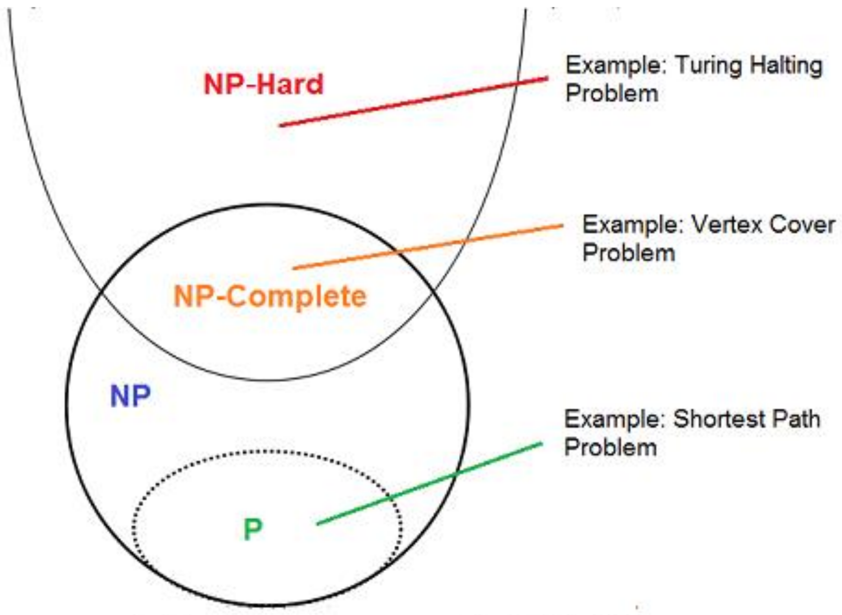
One of the central concepts in Computational Complexity Theory is the notion of "intractability." Intractability refers to problems for which no efficient algorithm exists—specifically, algorithms that can solve the problem in a time that grows polynomials with the size of the input. Problems that fall into this category are often considered difficult to solve for large inputs, and it's unlikely that we can find algorithms that can solve them quickly.

Computational Complexity Theory defines a hierarchy of complexity classes that categorize problems based on their computational difficulty. Here are some of the most important complexity classes:

1. P (Polynomial Time): As mentioned earlier, P consists of decision problems that can be solved by a deterministic Turing machine in polynomial time. These are problems for which efficient algorithms exist.
2. NP (Nondeterministic Polynomial Time): NP contains decision problems for which a proposed solution can be verified in polynomial time. The P vs. NP problem asks whether P = NP, meaning every problem that can be verified quickly can also be solved quickly.
3. NP-complete: A problem is NP-complete if it's in NP and every problem in NP can be polynomial-time reduced to it. NP-complete problems are believed to be some of the hardest problems in NP, and finding a polynomial-time algorithm for any NP-complete problem would imply polynomial-time solutions for all NP problems.
4. NP-hard: A problem is NP-hard if every problem in NP can be polynomial-time reduced to it. NP-hard problems might not be in NP themselves, but they are at least as hard as the hardest problems in NP.

These classes can be defined as P as Easy < NP as Medium < NP-Complete as Hard < NP-Hardest.




NP-Hard
Example: Turing Halting Problem
Example: Vertex Cover Problem
NP-Complete
NP
Example: Shortest Path Problem
P

This diagram assumes that P != NP

<u>P Class Problems (10 mins)</u>

In computational complexity theory, the class P (Polynomial Time) consists of decision problems that can be solved by a deterministic Turing machine in polynomial time. In simpler terms, problems in class P are those for which there exist algorithms that can solve them efficiently, with the runtime of the algorithm growing at most as a polynomial function of the size of the input.

Here are some key characteristics and examples of problems in the class P:

1. Efficient Algorithms: Problems in class P have efficient algorithms that can solve them in a reasonable amount of time. The running time of these algorithms is bounded by a polynomial function of the input size.
2. Polynomial Time: The algorithm's running time is proportional to a polynomial function of the input size. For example, if the input size is "n," the algorithm runs in time O(n^k), where "k" is a constant.
3. Deterministic Algorithms: Algorithms in P are deterministic, meaning that given the same input, they will produce the same output and follow the same sequence of steps every time.

Examples of problems in class P include:

1. Sorting: Given a list of numbers, arrange them in non-decreasing order. Algorithms like Merge Sort and Quick Sort run in O (n log n) time, making this problem a member of P.
2. Searching: Given a sorted list of numbers and a target number, determine if the target number is present in the list. Algorithms like Binary Search run in O (log n) time, which is a polynomial time complexity.
3. Shortest Path: Given a weighted graph, find the shortest path between two specified nodes. Algorithms like Dijkstra's algorithm and the Bellman-Ford algorithm have polynomial time complexity for this problem.
4. Matrix Multiplication: Multiply two matrices of appropriate dimensions. Algorithms like the Strassen algorithm and the more general Coppersmith-Winograd algorithm have polynomial time complexity for matrix multiplication.
5. Primality Testing: Determine whether a given number is prime. Algorithms like the Miller-Rabin primality test can determine primality in polynomial time.

<u>NP Class Problems (10 mins)</u>

In computational complexity theory, the class NP (Nondeterministic Polynomial Time) consists of decision problems for which a proposed solution can be verified in polynomial time. In other words, if someone claims to have a solution to an NP problem, you can efficiently check whether their solution is correct or not. The "Nondeterministic" aspect of NP refers to the theoretical concept of a nondeterministic Turing machine, which can explore multiple paths of computation simultaneously.

Here are some key characteristics and examples of problems in the class NP:

1. Nondeterministic Verification: For an NP problem, if someone provides a potential solution, you can verify its correctness in polynomial time using a deterministic algorithm. However, finding the solution itself might not be as straightforward.
2. Polynomial Time Verification: The verification algorithm runs in polynomial time with respect to the size of the input and the length of the solution.
3. No Requirement for Finding Solutions: NP problems don't necessarily require that you can find a solution efficiently. They only require that given a solution, you can verify it efficiently.

Examples of problems in class NP include:

1. Traveling Salesman Problem (TSP): Given a list of cities and distances between them, find the shortest route that visits each city exactly once and returns to the starting city. Verifying a proposed solution (a specific route) can be done by checking that it visits each city once and its total distance is within a certain bound.
2. Boolean Satisfiability Problem (SAT): Given a Boolean formula, determine if there exists an assignment of truth values to its variables that makes the formula true. Verifying a solution involves substituting the proposed truth values into the formula and checking if it evaluates to true.
3. Clique Problem: Given an undirected graph, determine if there exists a subset of vertices (a clique) such that every pair of vertices in the subset is connected by an edge. Verifying a proposed clique involves checking that all pairs of vertices are connected.
4. Subset Sum Problem: Given a set of integers and a target sum, determine if there exists a subset of the integers that adds up to the target sum. Verifying a proposed subset involves checking if the selected integers sum to the desired target.

NP-Complete Problems (10 mins)

NP-complete problems are a subset of problems within the NP (Nondeterministic Polynomial Time) complexity class that have a special property. A problem is classified as NP-complete if it is both in NP and is at least as "hard" as the hardest problems in NP with respect to polynomial-time reductions. In other words, an NP-complete problem is a problem to which any problem in NP can be efficiently reduced while preserving the structure of the problem.

The concept of NP-completeness was introduced by Stephen Cook and Leonid Levin in the 1970s, and it has had a profound impact on the understanding of computational complexity. The NP-completeness of a problem means that if you find a polynomial-time algorithm for solving it, you would effectively have a polynomial-time algorithm for solving all problems in NP.

Here are some key properties and examples of NP-complete problems:

1. Hardness: NP-complete problems are considered among the most difficult problems in NP. If you can solve one NP-complete problem efficiently, you can solve all problems in NP efficiently.
2. Reductions: NP-complete problems are related by polynomial-time reductions. If you can reduce problem A to problem B in polynomial time and you know problem B is NP-complete, then problem A is also at least as hard as problem B.
3. Clique to Vertex Cover: An example of a common NP-complete problem is the Vertex Cover problem. Given an undirected graph and an integer k, the goal is to find a set of at most k vertices such that every edge in the graph is incident to at least one vertex in the set. The Clique problem is another NP-complete problem where you're asked to find a subset of vertices that forms a complete subgraph (clique) of a given size. These two problems are related by a reduction: if you can solve Vertex Cover efficiently, you can solve Clique efficiently, and vice versa.
4. SAT to 3-SAT: The Boolean Satisfiability problem (SAT) involves determining if a given Boolean formula can be satisfied by assigning truth values to its variables. 3-SAT is a variant where each clause contains exactly three literals (variables or their negations). 3-SAT is NP-complete, and it's often used as a basis for proving the NP-completeness of other problems.
5. Traveling Salesman Problem (TSP): The decision version of TSP, where you need to determine whether there's a Hamiltonian cycle (a cycle that visits every vertex exactly once) with a total length less than or equal to a given bound, is also NP-complete.
6. The discovery of NP-completeness led to the development of the notion of "reductions" and a deeper understanding of the hierarchy of complexity classes. Solving an NP-complete problem efficiently remains an open question, and whether P = NP or not is one of the most famous unsolved problems in computer science.

NP Hard (5 mins)

NP-hard problems are a class of computational problems that are at least as difficult as the hardest problems in the NP (Nondeterministic Polynomial Time) complexity class. Unlike NP-complete problems, NP-hard problems don't necessarily need to be in NP themselves. They can be even more difficult and might not have polynomial-time verification algorithms. In essence, an NP-hard problem is one for which solving it efficiently would allow you to solve all problems in NP efficiently. NP-complete problems are a subset of NP-hard problems.

Here are some key characteristics and examples of NP-hard problems:

1. Hardness: NP-hard problems are considered among the most challenging problems in computational complexity theory. They are not required to have polynomial-time verification algorithms, making them potentially more difficult than NP-complete problems.
2. Reductions: Just like with NP-complete problems, NP-hard problems are related by polynomial-time reductions. If you can solve an NP-hard problem efficiently, you can solve all problems in NP efficiently.
3. Variety of Domains: NP-hard problems can be found in various domains, including optimization, graph theory, scheduling, and more.

Examples of NP Hard: Shortest path and Optimization path.

Video References (15 Min)

1. P vs. NP and the Computational Complexity Zoo
   https://youtu.be/YX40hbAHx3s?si=oMRtOPXyfNobC1F_

2. What is a polynomial-time reduction? (NP-Hard + NP-Complete) by Easy Theory
   https://youtu.be/O7pq43hIE_0?si=Isbcbu4gdq8quEHI