Lecture 18: P & NP

CLRS, pp.966-982

The course so far: techniques for designing efficient algorithms, e.g., divide-and-conquer, dynamic-programming, greedy-algorithms.

What happens if you can't find an efficient algorithm? Is it your "fault" or the problem's?

Showing that a problem has an efficient algorithm is, relatively, easy. "All' that is needed is to demonstrate an algorithm.

Proving that no efficient algorithm exists for a particular problem is difficult. How can we prove the non-existence of something?

We will now learn about NP Complete Problems, which provide us with a way to approach this question.

NP-Complete Problems

This is a very large class of thousands of practical problems for which

- it is not known if the problems have “efficient” solutions
- it is known that if *any one* of the NP-Complete Problems has an efficient solution then *all* of the NP-Complete Problems have efficient solutions
- researchers have spent innumerable man-years trying to find efficient solutions to these problems and failing
- there is a large body of tools that often permit us to prove when a new problem is NP-complete.
- The problem of finding an efficient solution to an NP-Complete problem is known, in shorthand as $P \neq NP$**?** . There is currently a US$1,000,000 award offered by the Clay Institute (*http://www.claymath.org/*) for its solution.

In the remainder of the course we will introduce the notation and terminology needed to properly discuss NP-Complete problems and the tools required to prove that problems are NP-complete.

Proving that a problem is NP-Complete does not prove that the problem is hard. It does indicate that the problem is very likely to be hard.

Time permitting, we will also discuss what to do if you find that the problem that you really need to solve is NP-complete (other than giving up).

Contents of this Lecture

In this lecture we introduce the concepts that will permit us to discuss whether a problem is ‘hard’ or ‘easy’. $\mathcal{NP}$-Complete problems themselves will not be introduced until the next-lecture.

- Input size of problems.
  Formalizes the idea of input size of problems to be the number of bits required to encode the problem. We also see where we need to be precise and where we don’t.

- Optimization problems vs. decision problems.
  Decision Problems have Yes/No answers.
  Optimization Problems require answers that are optimal configurations.
  Decision problems are “easier” than optimization problems; if we can show that a decision problem is hard that will imply that its corresponding optimization problem is also hard.

- Polynomial time algorithms. The Class $\mathcal{P}$.

- The Class $\mathcal{NP}$.

- Problems in the two classes.

- The class Co$-\mathcal{NP}$.

Encoding the Inputs of Problems

In order to formally discuss how hard a problem is, i.e., how much time it requires to solve as a function of its input, we need to be much more formal than before about the input size of a problem. We will therefore spend some time now discussing how to encode the inputs of problems.

Example: How do we encode graphs?

A graph G may be represented by its adjacency matrix $A = [a_{ij}]$. G can then be encoded as the binary string

$$a_{11} \cdots a_{1n} a_{21} \cdots a_{2n} \cdots a_{n1} \cdots a_{nn}$$

of length n^2.

When the binary string is given, the computer can count the number of bits and then determine n, the vertices and the edges.

Remark: In general, the inputs of any problem can be encoded as binary strings.

The Input Size of Problems

The input size of a problem may be defined in a number of ways.

Standard Definition: The input size of a problem is the minimum number of bits ($\{0, 1\}$) needed to encode the input of the problem.

Remark: The exact input size s, (minimal number of bits) determined by an optimal encoding method, is hard to compute in most cases. However, for the complexity problems we will study, we do not need to determine s exactly. See page 10 of these slides for a more precise description.

For most problems, it is suffi cient to choose some natural and (usually) simple, encoding and use the size s of this encoding.

Input Size Example: *Composite*

Problem: Given a positive integer n, are there integers $j, k > 1$ such that $n = jk$? (i.e., is n a composite number?)

Question:
What is the input size of this problem?

Answer: Any integer $n > 0$ can be represented in the binary number system as:

$$n = \sum_{i=0}^{k} a_i 2^i \quad \text{where } k = \lceil \log_2(n+1) \rceil - 1$$

and so be represented by the string $a_0 a_1 \ldots a_k$ of length $\lceil \log_2(n+1) \rceil$.

Therefore, a natural measure of input size is $\lceil \log_2(n+1) \rceil$ (or just $\log_2 n$).

Input Size Example: Sorting

Sorting problem: Sort n integers $a_1, \ldots, a_n$.

Question:
What is the input size of this problem?

Solution: Using fixed length encoding writes a_i as binary string of length

$$m = \lceil \log_2 \max(|a_i| + 1) \rceil.$$

This coding gives input size nm.

$\boxed{\text{Warning}}$

Running times of algorithms, unless otherwise specified, should be expressed in terms of input size.

For example, the naive algorithm for determining whether n is composite compares n against the first $n-1$ numbers to see if any of them divides n. This makes $\Theta(n)$ comparisons so it might seem linear and very efficient.

But, note that the size of the problem is $size(n) = \log_2 n$ so the number of comparisons performed is actually $\Theta(n) = \Theta\left(2^{size(n)}\right)$ which is exponential and not very good.

Input Size of Problems

Two positive functions $f(n)$ and $g(n)$ are of the same type if

$$c_1 g(n^{a_1})^{b_1} \le f(n) \le c_2 g(n^{a_2})^{b_2}$$

for all large n, where $a_1, b_1, c_1, a_2, b_2, c_2$ are some positive constants.

For example, all polynomials are of the same type, but polynomials and exponentials are of different types.

Suppose s is the actual input size in bits needed to encode the problem. From this point of view, any quantity t, satisfying

$$s^{a_1} \le t \le s^{a_2}$$

for some positive constants a_1 and a_2 (independent of s), may also be used as a measure of the input size of a problem.

This will simplify our discussions.

Input Size Example: Graphs

Graph problems: For many graph problems , the input is a graph $G = (V, E)$. What is the input size?

A natural choice: There are n vertices and e edges. So we need to encode $n+e$ objects. With fixed length coding, the input size is

$$(n + e)\lceil \log_2(n + e + 1)\rceil.$$

Since

$$[(n+e)\lceil \log_2(n+e+1)\rceil]^{1/2} < n+e < (n+e)\lceil \log_2(n+e+1)\rceil,$$

we may use $n + e$ as the input size.

$\boxed{\text{Input Size Example: Integer Multiplication}}$

Integer multiplication problem:
Compute $a \times b$.
What is the input size?

Solution: The (minimum) input size is

$$s = \lceil \log_2(a+1) \rceil + \lceil \log_2(b+1) \rceil .$$

A natural choice is to use

$$t = \log_2 \max(a, b)$$

as the input size since

$$\frac{s}{2} \le t \le s.$$

Decision Problems

Definition: A *decision problem* is a question that has two possible answers, yes and no.

Note: If L is the problem and x is the input we will often write $x \in L$ to denote a yes answer and $y \notin L$ to denote a no answer. Note: This notation comes from thinking of L as a *language* and asking whether x is in the language L (yes) or not (no). See CLRS, pp. 975-977 for more details

Definition: An *optimization problem* requires an answer that is an optimal configuration.

Remark: An optimization problem usually has a corresponding decision problem.

Examples that we will see:
MST vs. Decision Spanning Tree (DST)
Knapsack vs. Decision Knapsack (DKnapsack)
SubSet Sum vs. Decision Subset Sum (DSubset Sum)

Decision Problem: MST

Optimization problem: Minimum Spanning Tree

Given a weighted graph G, find a minimum spanning tree (MST) of G.

Decision problem: Decision Spanning Tree (DST)

Given a weighted graph G and an integer k, does G have a spanning tree of weight at most k?

The inputs are of the form (G, k). So we will write $(G, k) \in DST$ or $(G, k) \notin DST$ to denote, respectively, yes and no answers.

Decision Problem: Knapsack

We have a knapsack of capacity W (a positive integer) and n objects with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$, where v_i and w_i are positive integers.

Optimization problem: Knapsack

Find the largest value $\sum_{i \in T} v_i$ of any subset that fits in the knapsack, that is, $\sum_{i \in T} w_i \leq W$.

Decision problem: Decision Knapsack (DKnapsack)

Given k, is there a subset of the objects that fits in the knapsack and has total value at least k?

$\boxed{\text{Decision Problem: Subset Sum}}$

The input is a positive integer C and n objects whose values are positive integers $s_1, ..., s_n$. (For a more formal definition see CLRS, Section 34.4.5)

Optimization problem: Subset Sum

Among subsets of the objects with sum at most C, what is the largest subset sum?

Decision problem: Decision Subset Sum(DSubset Sum)

Is there a subset of objects whose values add up to exactly C?

Optimization and Decision Problems

- For almost all optimization problems there exists a corresponding simpler decision problem.

- Given a subroutine for solving the optimization problem, solving the corresponding decision problem is usually be trivial.
  Example: If we know how to solve MST we can solve DST which asks if there is an Spanning Tree with weight at most k. How? First solve the MST problem and then check if the MST has cost $\leq k$. If it does, answer Yes. If it doesn't, answer No.

- Thus if we prove that a given decision problem is hard to solve efficiently, then it is obvious that the optimization problem must be (at least as) hard.

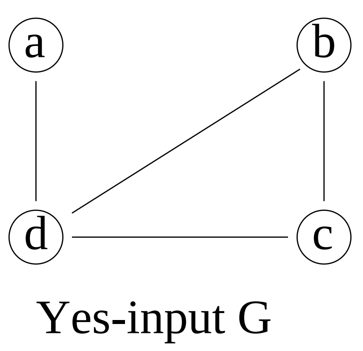
Note: The reason for introducing Decision problems is that it will be more convenient to compare the 'hardness' of decision problems than of optimization problems (since all decision problems share the same form of output, either yes or no.)

Decision Problems: Yes-Inputs and No-Inputs

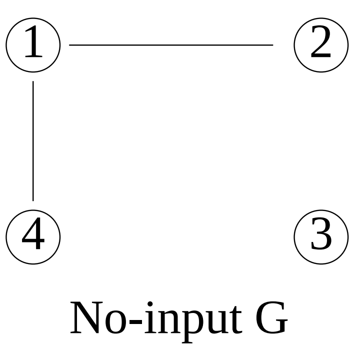
Yes-Input and No-Input: An instance of a decision problem is called a yes-input (resp. no-input) if the answer to the instance is yes (resp. no).

CYC Problem: Does an undirected graph G have a cycle?

Example of Yes-Inputs and No-Inputs:



Yes-input G



No-input G

Decision Problems: Yes-Inputs and No-Inputs

Decision Problem (TRIPLE):

Does a triple (n, e, t) of nonnegative integers satisfy $n - e = t$?

Example of Yes-Inputs: $(9, 7, 2)$, $(20, 2, 18)$.

Example of No-Inputs: $(10, 1, 2)$, $(20, 5, 18)$.

Complementary Problems

Let L denote some decision problem. The complementary problem $\bar{L}$ is the decision problem such that the yes-answers of $\bar{L}$ are exactly the no-answers of L. Note that $\bar{\bar{L}} = L$.

Example:
COMPOSITE: is given positive integer n composite (that is, can it be factored as $n = ab$, where $1 < a \leq b < n$)?
PRIMES: is given positive integer n a prime number?

$$\begin{aligned}\overline{\text{COMPOSITE}} &= \text{PRIMES}.\\ \overline{\text{PRIMES}} &= \text{COMPOSITE}\end{aligned}$$

Complexity Classes

The Theory of Complexity deals with

- the classification of certain
  “decision problems” into several classes:
  the class of “easy” problems,
  the class of “hard” problems,
  the class of “hardest” problems;

- relations among the three classes;

- properties of problems in the three classes.

Question: How to classify decision problems?

Answer: Use “polynomial-time algorithms.”

Polynomial-Time Algorithms

Definition: An algorithm is *polynomial-time* if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

Remark: Whether you use n or n^a (for fixed $a > 0$) as the input size, it will not affect the conclusion of whether an algorithm is polynomial time.

This explains why we introduced the concept of two functions being of the same type earlier on. Using the definition of polynomial-time it is not necessary to fixate on the input size as being the exact minimum number of bits needed to encode the input!

Polynomial-Time Algorithms

Examples of Polynomial-Time Algorithms

- The standard multiplication algorithm learned in school has time $O(m_1 m_2)$ where m_1 and m_2 are, respectively, the number of digits in the two integers.

- DFS has time $O(n + e)$.

- Kruskal's MST algorithm runs in time $O((e+n)\log n)$.

$\boxed{\text{Nonpolynomial-Time Algorithms}}$

Definition: An algorithm is *non-polynomial-time* if the running time is not $O(n^k)$ for any fixed $k \geq 0$.

Example: Let's return to the brute force algorithm for determining whether a positive integer N is a prime: it checks, in time $\Theta((\log N)^2)$, whether K divides N for each K with $2 \leq K \leq N-1$. The complete algorithm therefore uses $\Theta(N(\log N)^2)$ time.

Conclusion: The algorithm is nonpolynomial! Why?
The input size is $n = \log_2 N$, and so

$$\Theta(N(\log N)^2) = \Theta(2^n n^2).$$

Is Knapsack Polynomial?

Recall the problem. We have a knapsack of capacity W (a positive integer) and n objects with weights $w_1, ..., w_n$ and values $v_1, ..., v_n$, where v_i and w_i are positive integers. The optimization problem is to fi nd the largest value $\sum_{i \in T} v_i$ of any subset that fi ts in the knapsack, that is, $\sum_{i \in T} w_i \leq W$. The decision problem is, given k, to fi nd if there is a subset of the objects that fi ts in the knapsack and has total value at least k?

In class we saw a $\Theta(nW)$ dynamic programming algorithm for soving the optimization version of Knapsack. Is this a polynomial algorithm?

No! The size of the input is

$$size(I) = \log_2 W + \sum_i \log_2 w_i + \sum_i \log_2 v_i.$$

nW is not polynomial in $size(I)$. Depending upon the values of the w_i and v_i, nW could even be exponential in $size(I)$.

It is unknown as to whether there exists a polynomial time algorithm for Knapsack. In fact, Knapsack is a $\mathcal{NP}$-Complete problem, so anyone who could determine whether there was a polynomial-time algorithm for solving it would be proving that $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$ and would win the $US\$1,000,000$ prize from the Clay Institute!

Polynomial- vs. Nonpolynomial-Time

- Nonpolynomial-time algorithms are *impractical*.
  For example, to run an algorithm of time complexity 2^n for $n = 100$ on a computer which does 1 Terraoperation (10^{12} operations) per second:
  It takes $2^{100}/10^{12} \approx 10^{18.1}$ seconds
  $\approx 4 \cdot 10^{10}$ years.

- For the sake of our discussion of complexity classes Polynomial-time algorithms are *"practical"*.
  Note: in reality an $O(n^{20})$ algorithm is not really practical.

Polynomial-Time Solvable Problems

Definition: A problem is *solvable in polynomial time* (or more simply, the problem is *in polynomial time*) if there exists an algorithm which solves the problem in polynomial time.

Examples: The integer multiplication problem, and the cycle detection problem for undirected graphs.

Remark: Polynomial-time solvable problems are also called *tractable* problems.

$\boxed{\text{The Class } \mathcal{P}}$

Definition: The class $\mathcal{P}$ consists of all decision problems that are solvable in polynomial time. That is, there exists an algorithm that will decide in polynomial time if any given input is a yes-input or a no-input.

How to prove that a decision problem is in $\mathcal{P}$?
You need to find a polynomial-time algorithm for this problem.

How to prove that a decision problem is not in $\mathcal{P}$?
You need to prove there is no polynomial-time algorithm for this problem (much harder).

The Class $\mathcal{P}$: An Example

Example problem:
Is a given connected graph G a tree?

This problem is in $\mathcal{P}$.
Proof: We need to show that this problem is solvable in polynomial time. We run DFS on G for cycle detection. If a back edge is seen, then output NO, and stop. Otherwise output YES.

Recall that the input size is $n + e$, and DFS has running time $O(n + e)$. So this algorithm is linear, and the problem is in $\mathcal{P}$.

The Class $\mathcal{P}$: Another Example

Example problem: DST.
Given weighted graph G and parameter $k > 0$ does G have a spanning tree with weight $\leq k$?

This problem is in $\mathcal{P}$.
Proof: Run Kruskal's algorithm and find a *minimal spanning tree*, T, of G. Calculate $w(T)$ the weight of T. If $k \leq w(T)$, answer Yes ; otherwise, answer No.

Recall that Kruskal's algorithm runs in $O((e+n)\log n)$ time so this is polynomial in the size of the input.

Certificates and Verifying Certificates

We have already seen the class $\mathcal{P}$. We are now almost ready to introduce the class $\mathcal{NP}$. Before doing so we must first introduce the concept of *Certificates*.

Observation: A decision problem is usually formulated as:

Is there an object satisfying some conditions?

A **Certificate** is a specific object corresponding to a yes-input, such that it can be used to show the validity of that yes-input.

By definition, only yes-input needs a certificate (a no-input does not need to have a ’certificate’ to show it is a no-input).

Verifying a certificate: Given a presumed yes-input and its corresponding certificate, by making use of the given certificate, we verify that the input is actually a yes-input.

$\boxed{\textbf{The Class } \mathcal{NP}}$

Definition: The class $\mathcal{NP}$ consists of all decision problems such that, for each yes-input, there exists a certificate that can be verified in polynomial time.

Remark: $\mathcal{NP}$ stands for “nondeterministic polynomial time”. The class $\mathcal{NP}$ was originally studied in the context of nondeterminism, here we use an equivalent notion of verifi cation.

$\boxed{\text{COMPOSITE} \in \mathcal{NP}}$

COMPOSITE: Is given positive integer n composite? For COMPOSITE, an yes-input is just n, which means n is a composite.

Certificate: What is needed to show n (a presumed yes-input) is actually a yes-input? This is the certificate for COMPOSITE.

The certificate is an integer a ($1 < a < n$) with the property that it divides n.

Verifying a certificate: Given a certificate a, check whether a divides n. This can be done in time $O((\log_2 n)^2)$ (recall that input size is $\log_2 n$ so this is polynomial in input size).

Hence, COMPOSITE $\in \mathcal{NP}$.

$\boxed{\text{DSubsetSum} \in \mathcal{NP}}$

DSubsetSum: Input is a positive integer C and n positive integers $s_1, ..., s_n$. Is there a subset of these integers that add up to exactly C? A DSubsetSum yes-input consists of n numbers, and an integer C, which means there is a subset of those integers that add up to C.

Certificate: What is needed to show the given input is actually a yes-input? A subset T of subscripts (the corresponding integers should add up to C).

Verifying a certificate: Given a subset T of subscripts, check whether $\sum_{i \in T} s_i = C$.

Input-size is $m = (\log_2 C + \sum_{i=1}^{n} \log_2 s_i)$
and verification can be done in time

$$O(\log_2 C + \sum_{i \in T} \log_2 s_i) = O(m).$$

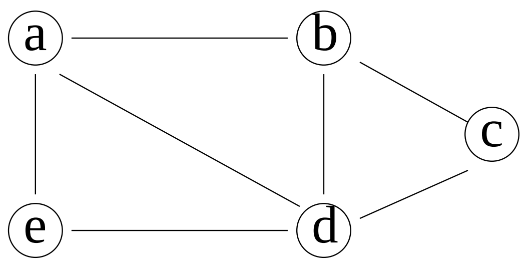
so this is polynomial time.

Hence we have DSubsetSum $\in \mathcal{NP}$.

$\boxed{\text{DHamCyc} \in \mathcal{NP}}$

Hamiltonian Cycle: Input is a graph $G = (V, E)$. A cycle of graph G is called *Hamiltonian* if it contains every vertex exactly once.

Example



Find a Hamiltonian cycle for this graph

Optimization problem: HamCyc

Find a Hamiltonian cycle for this graph or say that one doesn't exist.

Decision problem: DHamCyc

Does G have a Hamiltonian cycle?

$\boxed{\text{DHamCyc} \in \mathcal{NP}}$

Certificate: an ordering of the n vertices in G (corresponding to their order along the Hamiltonian Cycle), i.e., $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$.

Verification: Given a certificate the verification algorithm checks whether it is a Hamiltonian cycle of G by simply checking whether all of the edges

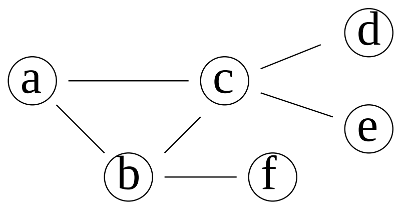
$$(v_{i_1}, v_{i_2}),\ (v_{i_2}, v_{i_3}), \ldots,\ (v_{i_{n-1}}, v_{i_n}),\ (v_{i_n}, v_{i_1})$$

appear in the graph. This can be done in $O(n)$ time so this is polynomial. Hence, DHamCyc $\in \mathcal{NP}$.

Important Note: There are many possible types of certifi cates for DHamCyc. This is only one such. Another type of certifi cate might be a set of n edges for which it has to be confi rmed that they are all in G and that they form a Hamiltonian Cycle.

$\boxed{\text{DVC} \in \mathcal{NP}}$

Vertex Cover: A vertex cover of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices.



Find a vertex cover of G of size two

Decision Vertex Cover (DVC) Problem: Given an undirected graph G and an integer k, does G have a vertex cover with k vertices.

Claim: DVC $\in \mathcal{NP}$.

Proof: A certificate will be a set C of $\leq k$ vertices. The brute force method to check whether C is a vertex cover takes time $O(ke)$. As $ke < (n+e)^2$, the time to verify is $O((n+e)^2)$. So a certificate can be verified in polynomial time.

Satisfiability I

We will now introduce Satisfiability (SAT), which, we will see later, is one of the most important $\mathcal{NP}$ problems.

Definition: A Boolean formula is a logical formula which consists of

boolean variables (0=false, 1=true),

logical operations

$\bar{x}$,	NOT,
$x \vee y$,	OR,
$x \wedge y$,	AND.

These are defined by:

x	y	$\bar{x}$	$x \vee y$	$x \wedge y$
0	0	1	0	0
0	1		1	0
1	0	0	1	0
1	1		1	1

Satisfiability II

A given Boolean formula is *satisfiable* if there is a way to assign truth values (0 or 1) to the variables such that the final result is 1.

Example: $f(x, y, z) = (x \wedge (y \vee \bar{z})) \vee (\bar{y} \wedge z \wedge \bar{x})$.

x	y	z	$(x \wedge (y \vee \bar{z}))$	$(\bar{y} \wedge z \wedge \bar{x})$	$f(x, y, z)$
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	0	1
1	0	1	0	0	0
1	1	0	1	0	1
1	1	1	1	0	1

For example, the assignment $x = 1$, $y = 1$, $z = 0$ makes $f(x, y, z)$ true, and hence it is satisfiable.

Satisfiability III

Example:

$$f(x,y) = (x \vee y) \wedge (\bar{x} \vee y) \wedge (x \vee \bar{y}) \wedge (\bar{x} \vee \bar{y}).$$

x	y	$x \vee y$	$\bar{x} \vee y$	$x \vee \bar{y}$	$\bar{x} \vee \bar{y}$	$f(x,y)$
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	0	1	1	0
1	1	1	1	1	0	0

There is no assignment that makes $f(x,y)$ true, and hence it is NOT satisfiable.

$\boxed{\text{SAT} \in \mathcal{NP}}$

SAT problem: Determine whether an input Boolean formula is satisfiable. If an Boolean formula is satisfiable, it is a yes-input; otherwise, it is a no-input.

Claim: SAT $\in \mathcal{NP}$.
Proof: The certificate consists of a particular 0 or 1 assignment to the variables. Given this assignment, we can evaluate the formula of length n (counting variables, operations, and parentheses), it requires at most n evaluations, each taking constant time. Hence, to check a certificate takes time $O(n)$. So we have SAT $\in \mathcal{NP}$.

$\boxed{k\text{-SAT} \in \mathcal{NP}}$

For a fixed k, consider Boolean formulas in k-conjunctive normal form (k-CNF):

$$f_1 \wedge f_2 \wedge \cdots \wedge f_n$$

where each f_i is of the form

$$f_i = y_{i,1} \vee y_{i,2} \vee \cdots \vee y_{i,k}$$

where each $y_{i,j}$ is a variable or the negation of a variable.

An example of a 3-CNF formula is

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee x_4).$$

k-SAT problem: Determine whether an input Boolean k-CNF formula is satisfiable.

Claim: 3-SAT $\in \mathcal{NP}$.
Claim: 2-SAT $\in \mathcal{P}$ (no proof given here).

Some Decision Problems in $\mathcal{NP}$

Some where we have given proofs:

Decision subset sum problem (DSubsetSum).
Decision Hamiltonian cycle (DHamCyc).
Satisfiability (SAT).
Decision vertex cover problem (DVC).

Some others (without proofs given; try to find proofs):

Decision minimum spanning tree problem (DMST).
Decision 0-1 knapsack problem (DKnapsack).

$\boxed{P = NP\textbf{?}}$

One of the most important problems in computer science is whether $P = NP$ or $P \neq NP$?

Observe that $P \subseteq NP$. Given a problem $\pi \in P$, and a certificate, to verify the validity of a yes-input (an instance of π), we can simply *solve* π in polynomial time (since $\pi \in P$). It implies $\pi \in NP$.

Intuitively, $NP \subseteq P$ is doubtful. After all, just able to *verify* a certificate (corresponds to a yes-input) in polynomial time does not necessary mean we can able to tell whether an input is an yes-input of no-input in polynomial time.

However, 30 years after the $P = NP$? problem was first proposed, we are still no closer to solving it and do not know the answer. The search for a solution, though, has provided us with deep insights into what distinguishes an “easy” problem from a “hard” one.

$\boxed{\text{The Class co-}\mathcal{NP}}$

Note that if $L \in \mathcal{NP}$, there is no guarantee that $\bar{L} \in \mathcal{NP}$(since having certificates for yes-inputs, does not mean that we have certificates for the no-inputs).

The class of decision problems L such that
$\bar{L} \in \mathcal{NP}$ is called co-$\mathcal{NP}$(observe it is does not require $L \in \mathcal{NP}$).

Example: $\text{COMPOSITE} \in \mathcal{NP}$ and so

$$\text{PRIMES} = \overline{\text{COMPOSITE}} \in \text{co}-\mathcal{NP}.$$

Remark: in contrast, to the fact that $L \in \mathcal{NP}$ does not necessarily imply $\bar{L} \in \mathcal{NP}$ we do have that

$L \in \mathcal{P}$, if and only if $\bar{L} \in \mathcal{P}$.

This is because a polynomial time algorithm for L is also a polynomial time algorithm for $\bar{L}$ (the NO-answers for L become Yes-answers for $\bar{L}$ and vice-versa).