COMP202
Complexity of Algorithms

$\mathcal{NP}$-completeness

[See Chapter 13 in Goodrich and Tamassia.]

Learning Outcomes

At the end of this series of notes, students should

1. Understand the fundamental importance of $\mathcal{NP}$-completeness in the theory of algorithmic computing and mathematics;
2. Comprehend the general theory in terms *efficient certificates* for verification and *reductions* used to demonstrate $\mathcal{NP}$-completeness;
3. Know some of the canonical $\mathcal{NP}$-complete problems such as the Hamilton Cycle Problem, 3-SAT, and $\{0, 1\}$ Integer Programming.

Hard Computational Problems

Some computational problems seem *hard* to solve.

Despite numerous attempts we do not *know* any *efficient* algorithms for these problems.

However, we are also far away from a *proof* that these problems are indeed hard to solve. We simply “don’t know” or are unsure right now.

Hard Computational Problems (cont.)

In more formal language, we don't know if $\mathcal{NP} = \mathcal{P}$ or $\mathcal{NP} \neq \mathcal{P}$.

This is an important question in theoretical computer science! And this set of lecture notes is designed to shed some light on this question.

As pointed out on the first set of lecture notes, this question is one of the seven Millennium Prizes set out by the Clay Institute, and can earn someone $ 1 million dollars for its solution (as well as a place in math/computer science history). See `http://www.claymath.org/millennium/P_vs_NP/` for their description of this problem.

Examples already encountered

We have already seen (at least) one example of these difficult problems where no efficient algorithm (i.e. polynomial-time) is known (i.e. one that will work on *all* inputs in polynomial-time).

This is the $\{0, 1\}$ Knapsack problem.

You have encountered another one in COMP 108 (I think), the Hamiltonian Cycle Problem.

The $\{0, 1\}$ Knapsack problem

The $\{0, 1\}$ Knapsack problem can be defined in the following way:

$\{0, 1\}$ Knapsack (optimization version)

Input: A collection of items $\{i_1, i_2, \ldots\ i_n\}$ where item i_j has integer weight $w_j > 0$ and gives integer benefit b_j. Also, a maximum weight W.

Goal: Find a subset of the items whose total weight does not exceed W and that maximizes the total benefit (taking fractional parts of items is *not allowed*).

The $\{0, 1\}$ Knapsack problem

The $\{0, 1\}$ Knapsack problem can be defined in the following way:

$\{0, 1\}$ Knapsack (optimization version)

Input: A collection of items $\{i_1, i_2, \ldots\ i_n\}$ where item i_j has integer weight $w_j > 0$ and gives integer benefit b_j. Also, a maximum weight W.

Goal: Find a subset of the items whose total weight does not exceed W and that maximizes the total benefit (taking fractional parts of items is *not allowed*).

Note: The dynamic programming algorithm for the $\{0, 1\}$ Knapsack problem runs in time $\theta(nW)$, which is *not* polynomial in the *size* of the problem (which would be something like $\theta(n \log W)$, i.e. using $\log W$ bits to represent the integer W).

Hamiltonian graphs

In COMP 108, you talked about Hamiltonian cycles in graphs.

More formally, we can define this *decision problem*:

Hamiltonian Cycle Problem

Input: A connected graph G.

Question: Does G have a Hamiltonian cycle, i.e. a cycle that passes through every vertex exactly once (except for the starting and ending vertex)?

Decision/Optimization problems

$\mathcal{NP}$-completeness is formulated in terms of decision problems.

Decision/Optimization problems

$\mathcal{NP}$-completeness is formulated in terms of decision problems.

A *decision problem* is a computational problem for which the output is either *yes* or *no*.

Decision/Optimization problems

$\mathcal{NP}$-completeness is formulated in terms of decision problems.

A *decision problem* is a computational problem for which the output is either *yes* or *no*.

In contrast to this, in an *optimization problem* we try to *maximize* or *minimize* some function.

Decision/Optimization problems

$\mathcal{NP}$-completeness is formulated in terms of decision problems.

A *decision problem* is a computational problem for which the output is either *yes* or *no*.

In contrast to this, in an *optimization problem* we try to *maximize* or *minimize* some function.

An optimization problem can often be turned into a decision problem by adding a parameter k, and then asking whether the value of the function (from the optimization problem) is *at most* or *at least* k.

If a decision problem is *hard*, then its related optimization version must also be *hard*.

Decision/Optimization problems (cont.)

Example:

***Optimization problem*:** Let G be a connected graph with integer weights on its edges.

Find the weight of a minimum spanning tree in G.

Decision/Optimization problems (cont.)

Example:

***Optimization problem*:** Let G be a connected graph with integer weights on its edges.

Find the weight of a minimum spanning tree in G.

***Decision problem*:** Let G be a connected graph with integer weights on its edges, and let k be an integer.

Does G have a spanning tree of weight at most k?

Decision/Optimization problems (cont.)

$\{0, 1\}$ Knapsack (decision version)

Input: A collection of items $\{i_1, i_2, \ldots\ i_n\}$ where item i_j has integer weight $w_j > 0$ and gives integer benefit $b_j > 0$. Also, a maximum weight W and an integer k.

Decision problem: Is there a subset of the items whose total weight does not exceed W, and whose total benefit is at least k?

Decision/Optimization problems (cont.)

Note that being able to efficiently *answer* decision questions allows us to efficiently *solve* the related optimization problem.

Decision/Optimization problems (cont.)

Note that being able to efficiently *answer* decision questions allows us to efficiently *solve* the related optimization problem.

We can use a type of "binary search" to find the optimal solution.

If $B = \sum_j b_j$ is the maximum possible benefit for the Knapsack Problem, we can first ask "Is there an allowable subset with benefit at least $\lfloor B/2 \rfloor$?" Then (depending on the answer) "Is there one with benefit at least $\lfloor B/4 \rfloor$ (or $\lfloor 3B/4 \rfloor$)?" Etc.

Formalities

To be precise, the ideas behind $\mathcal{NP}$-completeness involve mathematical definitions of things like

- *Turing machines*,
- *languages*,
- *encodings of problems*
- *accepting strings*,
- *etc.*

Formalities

To be precise, the ideas behind $\mathcal{NP}$-completeness involve mathematical definitions of things like

- *Turing machines*,
- *languages*,
- *encodings of problems*
- *accepting strings*,
- *etc.*

I will avoid most of these formalities and concentrate on the main high-level ideas here.

Complexity Class $\mathcal{P}$

The *complexity class* $\mathcal{P}$ is the set of all decision problems X that can be solved in *worst-case* polynomial time.

That is, there is an algorithm A that if the answer to a decision problem s is “yes”, then this can be determined in time $p(|s|)$, where $|s|$ is the *size* of s, and $p(\cdot)$ is a *polynomial*.

Complexity Class $\mathcal{P}$

The *complexity class* $\mathcal{P}$ is the set of all decision problems X that can be solved in *worst-case* polynomial time.

That is, there is an algorithm A that if the answer to a decision problem s is "yes", then this can be determined in time $p(|s|)$, where $|s|$ is the *size* of s, and $p(\cdot)$ is a *polynomial*.

The precise meaning of *size* is typically expressed in terms of the **binary representation** of integers (such as weights and benefits in the Knapsack Problem, edge weights or capacities for shortest paths or maximum flows, the number of vertices or edges of a graph, etc).

Complexity Class $\mathcal{P}$

The *complexity class* $\mathcal{P}$ is the set of all decision problems X that can be solved in *worst-case* polynomial time.

That is, there is an algorithm A that if the answer to a decision problem s is “yes”, then this can be determined in time $p(|s|)$, where $|s|$ is the *size* of s, and $p(\cdot)$ is a *polynomial*.

The precise meaning of *size* is typically expressed in terms of the **binary representation** of integers (such as weights and benefits in the Knapsack Problem, edge weights or capacities for shortest paths or maximum flows, the number of vertices or edges of a graph, etc).

Again, I will mostly avoid this precise distinction, but will point out some instances where it is important.

Complexity Class $\mathcal{P}$ (cont.)

Many, many problems are in the class $\mathcal{P}$.

Examples of problems that we have seen in this course or in COMP 108 include (decision versions of)

- minimum spanning tree,
- fractional knapsack,
- shortest paths in graphs with non-negative edge weights,
- maximum flows,
- Euler tours, and
- task scheduling.

Efficient certification

Another important idea when we discuss $\mathcal{NP}$-completeness is the idea of *certification*.

Efficient certification

Another important idea when we discuss $\mathcal{NP}$-completeness is the idea of *certification*.

Consider the Hamiltonian Cycle problem. Let G be a connected graph.

Suppose through some unspecified means (like good guessing, divine intervention, etc.), we find a *candidate* for a Hamiltonian circuit, i.e. a list of vertices and edges that we *think* might be a Hamiltonian cycle in G.

Efficient certification

Another important idea when we discuss $\mathcal{NP}$-completeness is the idea of *certification*.

Consider the Hamiltonian Cycle problem. Let G be a connected graph.

Suppose through some unspecified means (like good guessing, divine intervention, etc.), we find a *candidate* for a Hamiltonian circuit, i.e. a list of vertices and edges that we *think* might be a Hamiltonian cycle in G.

Then it is easy to *check* if this is indeed a Hamiltonian cycle. That is, we simply check that all the proposed edges exist in G, that we indeed have a cycle, and that we hit every vertex in G once.

Efficient certification

Another important idea when we discuss $\mathcal{NP}$-completeness is the idea of *certification*.

Consider the Hamiltonian Cycle problem. Let G be a connected graph.

Suppose through some unspecified means (like good guessing, divine intervention, etc.), we find a *candidate* for a Hamiltonian circuit, i.e. a list of vertices and edges that we *think* might be a Hamiltonian cycle in G.

Then it is easy to *check* if this is indeed a Hamiltonian cycle. That is, we simply check that all the proposed edges exist in G, that we indeed have a cycle, and that we hit every vertex in G once.

If the candidate solution is indeed a Hamiltonian cycle, then it *verifies* that the answer to the decision problem is "yes."

Efficient certification (cont.)

In a similar fashion, given a collection of weights, benefits, and parameters W and k for an instance of the $\{0, 1\}$ Knapsack Problem, if I propose a subset of these items, it is *easy* to *check* if those items have total weight at most W and if the total benefit is at least k.

Efficient certification (cont.)

In a similar fashion, given a collection of weights, benefits, and parameters W and k for an instance of the $\{0, 1\}$ Knapsack Problem, if I propose a subset of these items, it is *easy* to *check* if those items have total weight at most W and if the total benefit is at least k.

If both of those conditions are satisfied, then we say that the subset of items is a *certificate* for the decision problem, i.e. it *verifies* that the answer to the $\{0, 1\}$ Knapsack decision problem is “yes.”

Efficient certification (cont.)

In a similar fashion, given a collection of weights, benefits, and parameters W and k for an instance of the $\{0, 1\}$ Knapsack Problem, if I propose a subset of these items, it is *easy* to *check* if those items have total weight at most W and if the total benefit is at least k.

If both of those conditions are satisfied, then we say that the subset of items is a *certificate* for the decision problem, i.e. it *verifies* that the answer to the $\{0, 1\}$ Knapsack decision problem is "yes."

The idea of *efficient certification* is what is used to define the class of problems called $\mathcal{NP}$.

Complexity class $\mathcal{NP}$

The class $\mathcal{NP}$ consists of all decision problems for which there exists an *efficient certifier*.

Complexity class $\mathcal{NP}$

The class $\mathcal{NP}$ consists of all decision problems for which there exists an *efficient certifier*.

An *efficient certifier* for a decision problem X is an algorithm B that takes two input strings s and t. The string s is the input to the decision problem.

"Efficient" means that B is a polynomial-time algorithm, i.e. there is a polynomial function $q(\cdot)$ so that for every input s, then s has answer "yes" if and only if there exists a string t such that $|t| \leq q(|s|)$ and $B(s, t) =$ "yes."

Complexity class $\mathcal{NP}$

The class $\mathcal{NP}$ consists of all decision problems for which there exists an *efficient certifier*.

An *efficient certifier* for a decision problem X is an algorithm B that takes two input strings s and t. The string s is the input to the decision problem.

"Efficient" means that B is a polynomial-time algorithm, i.e. there is a polynomial function $q(\cdot)$ so that for every input s, then s has answer "yes" if and only if there exists a string t such that $|t| \leq q(|s|)$ and $B(s, t) =$ "yes."

We can think of the string t as being a "proof" that the answer to the decision problem is "yes" for the input string s.

Another (equivalent) definition

An algorithm that *chooses* (by a good guess) some number of *non-deterministic bits* during its execution is called a *non-deterministic algorithm*.

We say that an algorithm A *non-deterministically accepts* a problem s if there exists a *choice of non-deterministic bits* that ultimately leads to the answer "yes."

Complexity Class $\mathcal{NP}$ (the sequel)

The class $\mathcal{NP}$ is the set of decision problems X that can be *non-deterministically accepted* in polynomial time.

The idea is that a non-deterministic algorithm can “guess” a candidate solution. If it is indeed a solution to the decision problem, then the efficient certifier can verify that it is a solution in polynomial time.

Complexity Class $\mathcal{NP}$ (the sequel)

The class $\mathcal{NP}$ is the set of decision problems X that can be *non-deterministically accepted* in polynomial time.

The idea is that a non-deterministic algorithm can "guess" a candidate solution. If it is indeed a solution to the decision problem, then the efficient certifier can verify that it is a solution in polynomial time.

In the language of the previous definition, for an input string s, the non-deterministic algorithm can generate a (polynomial length) string t, then we use the algorithm B to check if $B(s, t) =$ "yes." (And, importantly, if the answer is "yes", then B will terminate in polynomial time with that output.)

Why $\mathcal{NP}$?

The search for a string t that will cause an efficient certifier to accept the input s is often viewed as a *non-deterministic search* over a set of possible proofs.

For this reason, $\mathcal{NP}$ was named as an acronym for "non-deterministic polynomial time."

$\{0,1\}$ Knapsack is in $\mathcal{NP}$

$\{0,1\}$ **Knapsack:** Let $I = \{i_1, i_2, \ldots\ i_n\}$ denote a collection of items where item i_j has integer weight $w_j > 0$ and gives integer benefit b_j. Also, we're given a maximum weight W and an integer k.

Question: Is there a subset of the items whose total weight does not exceed W, and whose total benefit is at least k?

$\{0, 1\}$ Knapsack is in $\mathcal{NP}$

$\{0, 1\}$ **Knapsack:** Let $I = \{i_1, i_2, \ldots\ i_n\}$ denote a collection of items where item i_j has integer weight $w_j > 0$ and gives integer benefit b_j. Also, we're given a maximum weight W and an integer k.

Question: Is there a subset of the items whose total weight does not exceed W, and whose total benefit is at least k?

A non-deterministic algorithm chooses a subset of items (randomly or according to some other scheme), then checks that the total weight isn't too large (i.e. more than W) and if the total benefit is at least k.

This checking is done in polynomial time (polynomial in the size of the inputs, i.e. n and the logarithms of w_i, b_i, and W).

Hamiltonian Cycle is in $\mathcal{NP}$

Hamiltonian Cycle: Given a connected graph G, does there exist a Hamiltonian Cycle that visits every vertex of v exactly once?

A *non-deterministic* algorithm chooses a cycle (represented by a sequence of non-deterministic bits) and checks deterministically whether this cycle is Hamiltonian.

$\mathcal{P}$ and $\mathcal{NP}$

Note that $\mathcal{P} \subseteq \mathcal{NP}$. Basically, if there is a polynomial-time algorithm A to solve a problem, then we can “ignore” any proposed solution t and return the answer that the algorithm A gives (i.e. whatever A returns, either “yes” or “no”, we give that same answer).

$\mathcal{P}$ and $\mathcal{NP}$

Note that $\mathcal{P} \subseteq \mathcal{NP}$. Basically, if there is a polynomial-time algorithm A to solve a problem, then we can "ignore" any proposed solution t and return the answer that the algorithm A gives (i.e. whatever A returns, either "yes" or "no", we give that same answer).

The (million dollar) question that mathematicians and computer scientists *don't know* the answer to is whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$.

$\mathcal{P}$ and $\mathcal{NP}$

Note that $\mathcal{P} \subseteq \mathcal{NP}$. Basically, if there is a polynomial-time algorithm A to solve a problem, then we can "ignore" any proposed solution t and return the answer that the algorithm A gives (i.e. whatever A returns, either "yes" or "no", we give that same answer).

The (million dollar) question that mathematicians and computer scientists *don't know* the answer to is whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$.

There is generally common *belief* that $\mathcal{P}$ and $\mathcal{NP}$ are different, i.e. there is some problem that is in $\mathcal{NP}$ but is not in $\mathcal{P}$.

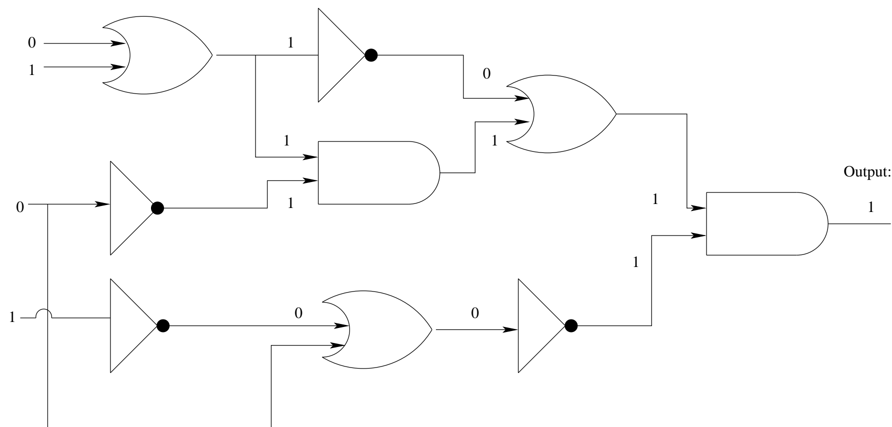
Boolean Circuits

In COMP109, you learned about propositional logic, and the basic operations for combining truth values. We use that here.

A *Boolean Circuit* is a *directed graph* where each node, called a *logic gate* corresponds to a simple Boolean function, one of *AND*, *OR*, or *NOT*.

The incoming edges for a logic gate correspond to inputs for its Boolean function and the outgoing edge corresponds to its output.

Boolean Circuit - Example



Circuit-SAT is in $\mathcal{NP}$

Circuit-SAT: Given a Boolean Circuit with a single output node, is there an assignment of values to the inputs so that the output value is 1?

A non-deterministic algorithm chooses an assignment of input bits and checks deterministically whether this input generates an output of 1.

The class co-$\mathcal{NP}$

The set of problems $\mathcal{NP}$ consists of those that have efficient verification methods for when the answer to the decision problem is "yes", but says nothing about the computation time for those problems having a "no" solution (indeed, it may be exceedingly difficult to verify the answer is "no" or there may not be a polynomial-time "no" verifier).

The class co-$\mathcal{NP}$

The set of problems $\mathcal{NP}$ consists of those that have efficient verification methods for when the answer to the decision problem is “yes”, but says nothing about the computation time for those problems having a “no” solution (indeed, it may be exceedingly difficult to verify the answer is “no” or there may not be a polynomial-time “no” verifier).

We can also think of the set of problems for which there is an efficient verification method for when the answer is “no”.

This is the class of problems that is called co-$\mathcal{NP}$.

The class co-$\mathcal{NP}$ (cont.)

A typical example of a problem in co-$\mathcal{NP}$ is “Given a positive integer n, is n a prime number?”

The class co-$\mathcal{NP}$ (cont.)

A typical example of a problem in co-$\mathcal{NP}$ is “Given a positive integer n, is n a prime number?”

If n is not prime, then an efficient way of verifying this fact is to give two integers k and ℓ (both bigger than 1) such that $k \cdot \ell = n$. (Multiplication is a polynomial-time operation in the size of the input numbers.)

Another problem...

Another question for which we don't know the answer is whether $\mathcal{NP} = \text{co-}\mathcal{NP}$.

Note: The set of problems in $\mathcal{P}$ lies in the intersection of $\mathcal{NP}$ and co-$\mathcal{NP}$.

Polynomial-time reducibility

Another important idea in the theory of $\mathcal{NP}$-completeness is the idea of transforming one problem instance into another problem instance. We formalize this idea below.

- We say that a problem L, defining some decision problem, is *polynomial-time reducible* to a problem M, if:
  there is a function f, computable in polynomial time, that takes an instance s of the problem L, and transforms it into an instance $f(s)$ of the problem M such that s has answer "yes" if and only if $f(s)$ has answer "yes".

Polynomial-time reducibility

Another important idea in the theory of $\mathcal{NP}$-completeness is the idea of transforming one problem instance into another problem instance. We formalize this idea below.

- We say that a problem L, defining some decision problem, is *polynomial-time reducible* to a problem M, if: there is a function f, computable in polynomial time, that takes an instance s of the problem L, and transforms it into an instance $f(s)$ of the problem M such that s has answer "yes" if and only if $f(s)$ has answer "yes".

In other words, we can transform an input for one decision problem into an appropriate input for another decision problem. Furthermore, the first problem has a "yes" solution if and only if the second decision problem has a "yes" solution.

We use notation $L \xrightarrow{poly} M$ to signify that problem L is polynomial time reducible to problem M.

$\mathcal{NP}$-hardness

We say that a problem M, defining some decision problem, is $\mathcal{NP}$*-hard* if every other problem L in $\mathcal{NP}$ is polynomial-time reducible to M. So this means that

- M is $\mathcal{NP}$-hard if for every $L \in \mathcal{NP}$, $L \xrightarrow{poly} M$.

If a language M is $\mathcal{NP}$-hard **and** it belongs to $\mathcal{NP}$ itself, then M is $\mathcal{NP}$*-complete*.

$\mathcal{NP}$-complete problems are some of the *hardest problems* in $\mathcal{NP}$, as far as polynomial reducibility is concerned.

Cook-Levin Theorem

The Cook-Levin Theorem states that *Circuit-SAT* is $\mathcal{NP}$-complete.

Proof idea: The computation steps of any (reasonable) algorithm can be simulated by layers in appropriately constructed (polynomial time and size) Boolean Circuit.

Other $\mathcal{NP}$-complete problems

We have just noted that there is *at least* one $\mathcal{NP}$-complete problem.

Using polynomial time reducibility we can show existence of other $\mathcal{NP}$-complete problems.

The following useful result also helps to prove $\mathcal{NP}$-completeness in many cases.

Lemma: If $L_1 \stackrel{poly}{\longrightarrow} L_2$ and $L_2 \stackrel{poly}{\longrightarrow} L_3$ then $L_1 \stackrel{poly}{\longrightarrow} L_3$.

Other $\mathcal{NP}$-complete problems (cont.)

Suppose that we have a new problem X and we think that X is $\mathcal{NP}$-complete. How can we do this?

- First, show that $X \in \mathcal{NP}$, i.e. show that X has a polynomial-time nondeterministic algorithm, or equivalently, show that X has an efficient certifier.
- Secondly, take a *known* $\mathcal{NP}$-complete problem Y, and demonstrate a polynomial time reduction from Y to X, i.e. show that $Y \stackrel{poly}{\longrightarrow} X$.

Types of reduction

Let M be an $\mathcal{NP}$-complete problem. Some types of reduction are:

- *Restriction*: Noting that a known $\mathcal{NP}$-complete problem is a special case of our problem L.
- *Local replacement*: Dividing instances of M and L into basic units, and then showing how each basic unit of M can be locally converted into a basic unit of L.
- *Component Design*: Building components for an instance of L that will enforce important structural functions on M.

$\mathcal{NP}$-complete problems we've seen already

- Hamiltonian cycle
- $\{0/1\}$ Knapsack
- Circuit-SAT

Conjunctive Normal Form

A Boolean formula is in *Conjunctive Normal Form* (CNF) if it is formed as a collection of *clauses* combined using the operator *AND* ($\wedge$), where each clause is formed by *literals* (variables x_i or their negations $\overline{x_i}$) combined using the operator *OR* ($\vee$).

Some examples are:

$$(\overline{x_1} \vee \overline{x_2} \vee x_4 \vee \overline{x_6}) \wedge (\overline{x_2} \vee x_4 \vee \overline{x_5} \vee x_3)$$

$$(x_2 \vee \overline{x_3} \vee \overline{x_1}) \wedge (x_1 \vee x_3 \vee x_4) \wedge (\overline{x_4} \vee \overline{x_2} \vee \overline{x_5})$$

$$(\overline{x_2} \vee x_1) \wedge (\overline{x_1} \vee x_4 \vee x_3) \wedge (x_4 \vee x_3) \wedge (\overline{x_3} \vee x_2 \vee \overline{x_5} \vee \overline{x_1})$$

CNF-SAT and 3-SAT

CNF-SAT
Input: A Boolean formula in CNF.

Question: Is there an assignment of Boolean values to its variables so that the formula evaluates to 1 (i.e. is the formula *satisfiable*)?

3-SAT is CNF-SAT in which each clause has exactly three literals.

Fact: CNF-SAT and 3-SAT are $\mathcal{NP}$-complete.

(Note: 2-SAT is in $\mathcal{P}$, i.e. there's a polynomial time algorithm when every clause has exactly *two* literals.)

Integer Programming

$\{0,1\}$ **Integer Programming:**

Input: A collection of inequalities of the form

$$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n \leq b_i$$

where the values $a_{i,j}, b_i$ are integers. Other inputs are integers $c_1, \ldots, c_n$ and an integer k.

Question: Is there a setting of the variables x_i so that each x_i is either 0 or 1, satisfying all of the inequalities, and such that

$$c_1x_1 + c_2x_2 + \cdots c_nx_n \geq k?$$

Subset Sum

Subset Sum:

Input: A set, S, of integers.

Question: Is there a non-empty subset $T \subseteq S$, such that the sum of the elements in T is zero?

3-Coloring of Graphs

3-COL:

Input: A graph G.

Question: Is G 3-colorable? In other words, is there an assignment of the labels $\{1, 2, 3\}$ to the *vertices* of G so that any two vertices that are adjacent are assigned *different* labels?

3-COL is part of the more general class of problems **k-COL**, where the vertices of a graph are assigned labels from the set $\{1, 2, \ldots, k\}$, and the goal is to assign labels so that adjacent vertices have different labels.

If $k \geq 3$, then **k-COL** is $\mathcal{NP}$-complete.