What is the goal of computational complexity theory?

- To provide a method of quantifying problem difficulty in an absolute sense.
- To provide a method for comparing the relative difficulty of two different problems.
- We would like to be able to rigorously define the meaning of efficient algorithm...
- ...and we would like to state that one algorithm is “better” than another.
- Complexity theory is built on a basic set of assumptions called the model of computation (e.g., Turing Machine). But, we will not concern ourselves too much with the details of a particular model here.

The ingredients that we need to build a theory of computational complexity for problem classification are the following...it’s a 4 ingredient recipe!

1. A class $C$ of problems to which the theory applies.
2. A (nonempty) subclass $C_E \subseteq C$ of “easy” problems.
3. A (nonempty) subclass $C_H \subseteq C$ of “hard” problems.
4. A relation $\prec$ “not more difficult than” between problems.

Our goal is just to put some theory around this machinery:

- $Q \in C_E$, $P \prec Q \implies P \in C_E$
- $P \in C_H$, $P \prec Q \implies Q \in C_H$

What is the difference between problem and (problem) instance?

- A problem is an infinite family of instances whose objective function and constraints have a specific structure.
How to measure the difficulty of an instance?

- The difficulty of a problem instance is easy to judge.
- Try to solve it and see how long it takes.
- Note that this inherently depends on the algorithm and the computing platform.
- We want a measure independent of both these variables.
- We will always assume the best known algorithm and best known implementation are used.
- We will measure execution time in terms of the total number of elementary operations executed (more on this later).
- We are interested in problems difficulty, rather than instances. Yet, the time needed to solve a problem instance with a given algorithm depends on certain properties of the instance.
- One such property is the size of the instance.

What is the size of an instance?

- In many cases, the size of an instance can be taken to be the number of input parameters.
- For a general MIP, this would be roughly determined by the number of variables and constraints.
- The running time of certain algorithms, however, depends explicitly on the magnitude of the input data.
- We will define the size of an instance to be the amount of information required to represent the instance.
- This is still not a clear definition because it depends on our “representation of the data” (the alphabet).
- Because computers store numbers in binary format, we use the size of a binary encoding (a two symbol alphabet) as our standard measure.
- In other words, the size of a number $n$ is the number of bits required to represent it in binary, i.e., $\lceil \log_2 n \rceil + 1$.
- As long as the magnitude of the input data is bounded, this is equivalent to considering the number of input parameters!
In practice, the magnitude of the input data is usually, but not always, bounded.

- Binary encoding is what computers do.
- An integer $2^n \leq x < 2^{n+1}$ can be represented by a vector $(\delta_0, \delta_1, \ldots, \delta_n)$, where $x = \sum_{i=0}^{n} \delta_i 2^n$.
- So, it requires a logarithmic number of bits to represent $x \in \mathbb{Z}$.
- Again, we always assume that numbers are rational, so they can be encoded with two integers.

Given a problem $P$ and its instances $X_1, X_2, \ldots$, let $l_i = L(X_i)$ be the binary string length of instance $X_i$. First, we observe that two instances of the same length may not have the same running time. Second, we would like to express the running time as a function of $l$. To achieve this, we must use some statistics to “aggregate” the running time of all instances of the same length.

- Possible methods of evaluation: best case running time, average case running time, worst case running time.
- Best case doesn’t give us any guarantee about the difficulty of a given instance.
- Average case is difficult to analyze and depends on specifying a probability distribution on the instances.
- Worst case addresses these problems and is usually easier to analyze.
- Note that these measures are still made relative to a particular algorithm.

**What do we mean by running time?**

- Running time is a measure of efficiency for an algorithm.
- In most cases, worst case running time depends primarily on the size of the instance, as we have defined it.
- Therefore, our measure will be the (worst case) running time required to solve a given instance as a function of the size of the instance.
- However, we still need a measure of running time that is architecture independent.
- We will simply count the number of elementary operations required to perform the algorithm.

**What do we mean by elementary operations?**
Elementary operations are very loosely defined to be additions, subtractions, multiplications, comparisons, etc.

In most cases, we will assume that each of these can be performed in constant time.

Again, this is a good assumption as long as the size of the numbers remains “small” as the calculation progresses.

Generally we will want to ensure that the numbers can be encoded in a size polynomial in the size of the input.

This justifies our assumption about constant time operations.

Example:

```plaintext
for i = 1...p do
  for j = 1...q do
    c[i,j] = a[i,j] + b[i,j]
```

1. How many elementary operations?
2. How long does an elementary operation take?
3. This may depend on the encoding! Anyway, all reasonable encodings would take at most on the order of $\log \theta$, where $\theta = \max_{i,j} \{a_{ij}, b_{ij}\}$
4. Again, we always assume that numbers are rational, so they can be encoded with two integers.
5. In what follows, assume all elementary operations (addition, multiplication, comparison, etc.) can be accomplished in constant time. In other words, assume that $\log \theta$ is negligible.

**Asymptotic Analysis**

So far, we have determined that our measure of running time will be a function of instance size (a positive integer).

Determining the exact function is still problematic at best.

We will only really be interested in approximately how quickly the function grows “in the limit”.

To determine this, we will use asymptotic analysis, aka “big $O$ notation”:

1. It provides a special way to compare relative sizes of functions.
2. It makes use of approximation that highlights “large scale” differences.

- So, let \( f \) and \( g \) be real-valued functions that are defined on the same set of real numbers.
- We say \( f \) is \( O(g(x)) \) iff there exist positive constants \( c \) and \( x_0 \) such that:
  \[
  f(x) \leq c \cdot g(x), \quad \forall \; x \geq x_0.
  \]
- In this case, we say \( f \) is “order” \( g \), or \( f \) is “big \( O \)” of \( g \), or also \( f \) and \( g \) are “of the same order”.
- Using this relation, we can divide functions into classes that are all of the same order.
- For polynomials, the order relation can be used to divide the set of functions into equivalence classes.
- We will only be concerned with what equivalence class the function belongs to.
- Note that class membership is invariant under multiplication by scalars and addition of “low-order” terms.
- For polynomials, the class is determined by the largest exponent on any of the variables.
- For example, all functions of the form \( f(n) = an^2 + bn + c \) are \( O(n^2) \).

1. We can prove that any polynomial function is “big \( O \)” of the polynomial that is the highest term.

2. Remember: polynomials are “bigger than” logarithms, and exponentials “are bigger” than polynomials.

### Running time and Complexity

- Running time is a measure of the efficiency of an algorithm.
- Computational complexity is a measure of the difficulty of a problem.
- The computational complexity of a problem is the running time of the best possible algorithm.
- In most cases, we cannot prove that the best known algorithm is also the best possible algorithm.
- We can therefore only provide an upper bound on the computational complexity in most cases.
- That is another reason why complexity is usually expressed using “big \( O \)” notation.
Comparing Algorithms

- So far, we have defined complexity as a tool for comparing the difficulty of two different problems.
- Clearly, this machinery can also be used to compare two algorithms for the same problem.
- In this way, we can judge whether one algorithm is “better” than another one.
- Note that worst case analysis is far from perfect for this job.
- The simplex algorithm has an exponential worst case running time, but does extremely well in practice.

Polynomial Time Algorithms

- Algorithms whose running time is bounded by a polynomial function are called *polynomial time algorithms*.
- For the purposes of this class, we will call an algorithm *efficient* if it is *polynomial* time.
- Problems for which a polynomial time algorithm exists are called polynomially solvable.
- The class of all problems which are known to be polynomially solvable occupies a special place in optimization theory.
- For most interesting problems, it is not known whether or not there is a polynomial algorithm.
- This is one of the great unsolved problems in mathematics.
- If you can solve it, the American Mathematical Society will give you one million dollars and you will become instantly famous.
So we are ready for (somewhat formal) definitions. Given a problem $P$, and algorithm $A$ that solves $P$, and an instance $X$ of problem $P$:

- $L(X)$ is the length (in a reasonable encoding) of the instance
- $f_A(X)$ is the number of elementary calculations required to run algorithm $A$ on instance $X$.
- $f^*_A(l) = \sup_X \{f_A(X) : L(X) = l\}$ is the running time of algorithm $A$.
- If $f^*_A(l) = O(l^p)$ for some positive constant integer $p$, $A$ is polynomial.
- $A$ is strongly polynomial if $f^*_A(l)$ is bounded by a polynomial function that does not involve the data size (magnitude of numbers).
- $A$ is weakly polynomial if it is polynomial and not strongly polynomial. I.e., the $l$ in $O(l^p)$ contains terms involving $\log \theta$.
- An algorithm is said to be an exponential-time algorithm if $f^*_A(l) \not= O(l^p)$, for any $p$.

The case of Linear Programming

- General linear programming is polynomially solvable.
- Note, however, that the simplex algorithm is not polynomial time!
- In practice, the expected running time is polynomial.
- A polynomial-time algorithm (the ellipsoid method) for LP was not found until 1979!
- Although this algorithm has not had a big practical impact, it’s theoretical impact has been large.
- This is one of the biggest cases against using worst-case analysis.

Pseudo-polynomial Time Algorithms

- A pseudo-polynomial algorithm is one that is polynomial in the length of the data when encoded in unary.
- Unary means that we are using a one-symbol alphabet.
- Hence, to store an integer $k$, we would need $k$ symbols.

Example: The Integer Knapsack Problem:
There is an $O(nb)$ algorithm for this problem, where $n$ is the number of items and $b$ is the size of the knapsack.

This is not a polynomial time algorithm in general.

If $b$ is bounded by a polynomial function of $n$, then it is.

However, it is pseudo-polynomial.

Let’s recap:

- To define “easy” and “hard”, we need to make a few definitions so we can define the running time of an algorithm.
- That’s because “easy” for us, is a problem for which we have an efficient algorithm to solve it and efficiency is expressed in terms of running time.
- Running time is expressed in terms of number of elementary operations...and
- ...the running time of an algorithm depends on the size of the input.
- A time complexity function specifies, as a function of the problem size, the largest (here is our “worst case”) amount of time needed by an algorithm to solve any problem instance (of that size).
- How do we measure problem size?
- The length of the amount of information necessary to represent the problem in a reasonable encoding scheme.
- So an algorithm is efficient if its running time is a polynomial function of the size of the instance.
- The class $C_E$ of “easy” problems consists of those problems for which we know a polynomial time algorithm to solve them.

So we have Ingredient #2 for our recipe!

So far, we miss three out of the four ingredients... We have defined Ingredient #2, i.e., the class of “easy problems” $C_E$. The class of “easy” problems are those for which we know an “efficient” algorithm to solve them (and “hard” problems are those for which such an algorithm does not exist). Hence, in order to define easy and hard classes, we had to define “efficient” algorithm. Efficient algorithm means polynomial time algorithm. Polynomial time means that the running time (in terms of number of operations) is polynomial in the size of the input. Problems for which such an algorithm exists, are “easy” problems, aka polynomially solvable.

**Feasibility Problems**
• A feasibility problem or decision problem $P$ is a pair $(D,F)$ with $F \subseteq D$, where the elements of $D$ are finite binary strings. $D$ is called the set of instances of $P$, and $F$ is called the set of feasible instances of $P$.

• Given an instance $d \in D$, we want to determine whether $d \in F$. Hence, a feasibility problem is a problem for which the answer is either “yes or no”.

• For technical reasons, most of complexity theory is defined in terms of decision problems.

• Associated with an optimization problem we define a feasibility problem in which an instance corresponds to a description of a constraint set. $F$ is the set of instances for which the constraint set is nonempty.

• A second feasibility problem concerns a lower bound on the objective function (assuming maximization problem). Here we augment each instance by an objective function $c$ and an integer $z$.

• The lower bound feasibility problem is the feasibility problem with the additional constraint $cx \geq z$.

• Any optimization problem is polynomially reducible to a decision problem:

  1. $\textbf{Opt}$: $\max\{c^T x \mid x \in S\}$
  2. $\textbf{Decision}$: $\exists x \in S \mid c^T x \geq k$?

• Example: The Bin Packing Problem (BPP)

  1. We are given a set $S$ of items, each with a specified integral size, and a specified constant $C$, the size of the bin.
  2. $\textbf{Opt}$: Determine the smallest number of subsets into which one can partition $S$ such that the total size of the items in each subset is at most $C$.
  3. $\textbf{Decision}$: For a given constant $K$ determine whether $S$ can be partitioned into $K$ subsets such that the total size of the items in each subset is at most $C$. 


Now, how can we turn \textbf{Opt} to \textbf{Decision}?

- Suppose you know that \( l \leq z^* \leq u \), \( l, z^*, u \in \mathbb{Z} \)
- \( z^* \) is optimal value of \textbf{Opt}
- How can you solve \textbf{Opt} by solving a series of \textbf{Decision} problems?
  1. for (\text{k}=1; \text{k}<=\text{u}; \text{k}++) dec(\text{k})
  2. Requires at most \( u - l + 1 \) calls to \text{dec}(\text{k})

- Better to use binary search:
  1. if(\text{u}-\text{l} <=1) \text{z} = \text{u}; exit();
  2. \text{k}=(\text{l}+\text{u})/2; if(dec(\text{k})==false) \text{l}=\text{k}; else \text{u}=\text{k}; goto 1;
  3. Requires at most \( \log(u - l) + 1 \) calls to \text{dec}(\text{k})
  4. The log is important! For reasons to be explained later.

\textbf{Nondeterministic Algorithms}

- Information that can be used to check feasibility in polynomial time is called a \textit{certificate of feasibility}.
- In other words, a certificate of feasibility for a decision problem is information that can be used to verify “yes” in polynomial time.

- Given \( P = (D, F) \), for each instance \( d \in F \) we let \( Q_d \) denote such a certificate.
- We know that if \( Q_d \) exists, it must be \textit{short}.
- One might imagine an algorithm that makes a large number of guesses in the hope of eventually guessing \( Q_d \).
- This leads to the concept of \textit{nondeterministic} algorithm.
- Such algorithms cannot be realized in practical computation.
- A nondeterministic algorithm works on a feasibility problem \((D, F)\) as follows.

- The input to the algorithm is an instance \( d \in D \).
- The algorithm has two stages:
  - \textit{Guessing Stage}: Randomly guess a string \( Q \).
- **Checking Stage**: Check whether \( Q \) can be used to verify the feasibility of \( d \). If so, output \( d \in F \). If not, there is no output.

- There are two properties required:
  - We require that if \( d \in F \), then there must exist a certificate \( Q_d \) that verifies the feasibility of \( d \).
  - The running time of the algorithm is the time it takes to check a certificate that verifies feasibility.

- In other words, we measure the work done by a *nondeterministic* algorithm only in the checking stage and only when the *checking stage* is given \( d \in F \) and a *short certificate* of feasibility.

**Nondeterministic polynomial time algorithms**

- These algorithms are called nondeterministic because the guessing stage is random.

- A nondeterministic polynomial time algorithm is one for which the running time is a polynomial in the size of the input.

- In other words, we say that the *nondeterministic algorithm is polynomial* if, for each \( d \in F \), its running time in the checking stage is a polynomial function of the length of the encoding of \( d \) for some \( Q_d \) for which it replies \( d \in F \).

- This also means that when \( d \in F \) there is a *short (polynomial) proof of feasibility*.

- The class of problems for which there exists a nondeterministic polynomial time algorithm is denoted \( \mathcal{NP} \).

- The essential property of problems in \( \mathcal{NP} \) is that for every feasible instance, there exists a short certificate that can be checked in polynomial time.
So we have Ingredient #1 of our recipe: the class $\mathcal{NP}$ to which our theory applies consists of decision problems that can be solved in polynomial time on a non-deterministic Turing machine.

- $\mathcal{NP} \neq \text{"Non-polynomial"!}$
- $\mathcal{NP}$ is the class of decision problems that can be solved in polynomial time on a non-deterministic Turing machine.
- $\mathcal{NP}$ is the class of decision problems with the property that for every instance for which the answer is “yes”, there is a short certificate.
- The certificate is your “proof” that what you are telling me is the truth.
- Example: 01-IP feasibility is in $\mathcal{NP}$.
- Example: Hamiltonian Circuit $\mathcal{NP}$.
- Example: The complement of 01-IP feasibility is not in $\mathcal{NP}$!!!
- Example: The complement of Hamiltonian Circuit is not in $\mathcal{NP}$!!!

The classes $\mathcal{P}$, $\mathcal{NP}$, and $\text{CoNP}$

- $\text{CoNP}$ is the class of problems for which the complement is in $\mathcal{NP}$.
- $\mathcal{P}$ is the class of problems for which there exists a polynomial algorithm.
- Obviously, $\mathcal{P}$ is a subset of $\mathcal{NP}$.
- $\mathcal{P}$ is also a subset of $\text{CoNP}$. Really?
- Yes, every polynomial algorithm is also a nondeterministic polynomial algorithm. Ignore the guessing stage and apply the polynomial algorithm in the checking stage: $P \in \mathcal{P} \implies P \in \mathcal{NP}$. But if $P \in \mathcal{P}$, so is $\bar{P}$, since if $d \notin F$, it follows that our polynomial algorithm, which needs no guess, will also tell this in polynomial time. Hence, $\bar{P}$ is in $\mathcal{P}$, or equivalently $P \in \text{CoNP}$.
- So, $\mathcal{P} \subset (\mathcal{NP} \cap \text{CoNP})$.
- It is not known if $\mathcal{P} = \mathcal{NP} \cap \text{CoNP}$.
- It is not known if $\mathcal{NP} = \text{CoNP}$, but it’s unlikely.
• Also, it is unlikely that there exist many problems in \( NP \cap \text{CoNP} \) that are not also in \( P \).

• (Finally, it is not known whether \( P = NP \): the million dollar question!)

Where are we now? How many ingredients so far?

• We have a reference class \( C \) of problems consisting of \( NP \) problems
• We know a class of “easy” problems: \( P \)
• We still need a class of “hard” problems
• We still need a relation “not more difficult than”: \( \preceq \)

Feasibility and Polynomial Transformation

• Suppose we have two problems \( P_1 = (D_1, F_1) \) and \( P_2 = (D_2, F_2) \).

• In addition, we have a function \( g : D_1 \to D_2 \) such that : for \( d \in D_1 \), \( g(d) \in F_2 \iff d \in F_1 \).

• \( g(d) \) is computable in time polynomial in the length of \( d \).

• In this case, we say that \( P_1 \) is polynomially transformable to \( P_2 \).

• So, if problems \( P, Q \in NP \), and if any instance of \( P \) can be converted in polynomial time to an instance of \( Q \), then \( P \) is polynomially transformable to \( Q \).

• This is the “not more difficult than” relation that we needed!

• We will write this as \( P \preceq Q \).

Just one more ingredient and we are done!

The Class \( \text{NP} \cap \text{CoNP} \)

• It is interesting to ask what are the “hardest” problems in \( \text{NP} \cap \text{CoNP} \)?

• We say that a problem \( P \) is in the class \( \text{NP} \cap \text{CoNP} \) (stands for \( \text{NP-complete} \)) if every problem in \( \text{NP} \) is polynomially reducible to \( P \).

• Surprisingly, such problems exist!

• Even more surprisingly, this class contains almost every interesting integer programming problem that is not known to be in \( P \)!

• SAT is the first problem proven to be \( \text{NP-complete} \).
• After SAT (satisfiability problem) was proven to be $\mathcal{NP}$-complete, many other problems were proven to be $\mathcal{NP}$-complete.

• This was done by polynomial transformation.

Recap of the last ingredient:

• We asked the question: what are the hardest problems in $\mathcal{NP}$?

• We call this class of problems $\mathcal{NP}C$, i.e., $\mathcal{NP}$-complete

• Using the definitions we have made, we would like to say that if $P \in \mathcal{NP}C$, and $Q \in \mathcal{NP}$ $\implies$ $Q \triangleleft P$

• In other words, if $P \in \mathcal{NP}$ and we can convert in polynomial time every other problem $Q \in \mathcal{NP}$ to $P$, then $P$ is in this sense the “hardest” problem in $\mathcal{NP}$. So we say $P \in \mathcal{NP}C$.

• $Q \in \mathcal{P}, P \triangleleft Q \implies P \in \mathcal{P}$

• $P \in \mathcal{NP}C, P \triangleleft Q \implies Q \in \mathcal{NP}C$

The SAT problem

• This is the first problem proven to be $\mathcal{NP}C$.

• The problem is described by:

1. A finite set $N = \{1, \ldots, n\}$ (the literals)
2. $m$ pairs of subsets of $N$, $C_i = (C_i^+, C_i^-)$ (the clauses)
3. An instance is feasible if the set:

$$\{x \in \mathbb{B} \mid \sum_{j \in C_i^+} x_j + \sum_{j \in C_i^-} (1 - x_j) \geq 1 \quad i = 1, \ldots, m\}$$

is nonempty.

• This problem is obviously in $\mathcal{NP}$ (why?).

• In 1971, Cook defined the class $\mathcal{NP}$ and showed that satisfiability was $\mathcal{NP}C$, even if each clause only contains three literals.

• The proof is beyond the scope of this course.

• Important: since we have “described” SAT as 01-IP, then: 01-IP feasibility is $\mathcal{NP}$-complete.

The Class $\mathcal{NP}$-Hard
• Just one more class..be patient!
• The class $\mathcal{NP}$-Hard extends $\mathcal{NP}$-complete to include problems that are not in $\mathcal{NP}$.
• If $X_1 \in \mathcal{NP}$ and $X_1$ reduces to $X_2$, then $X_2$ is said to be $\mathcal{NP}$-Hard.
• Thus, all $\mathcal{NP}$-complete problems are $\mathcal{NP}$-Hard.
• The primary reason for this definition is so we can classify optimization problems that are not in $\mathcal{NP}$.
• It is common for people to refer to optimization problems as being $\mathcal{NP}$-complete, but this is technically incorrect.
• Generally speaking, most interesting problems are either known to be in $\mathcal{P}$ or are $\mathcal{NP}$-complete.
• The problems known to be in $\mathcal{P}$ are generally “easy” to solve.
• The problems in $\mathcal{NP}$-complete are generally “hard” to solve.
• In practice, it is true that most problem known to be in $\mathcal{P}$ are “easy” to solve.
• This is because most known polynomial time algorithms are of relatively low order.
• It seems very unlikely that $\mathcal{P} = \mathcal{NP}$.
• If so, the reduction is likely to be prohibitively expensive.
• For similar reasons, although all $\mathcal{NP}$C problems are “equivalent” in theory, they are not in practice. E.g., TSP vs. QAP

**Stable Set Problem is $\mathcal{NP}$-complete**

• A stable set $I$ in a graph $G = (V, E)$ is a subset of vertices $I \subseteq V$, no two of which are adjacent.
• Let’s now prove that the lower bound feasibility problem for stable set is $\mathcal{NP}$-complete.
• First, the problem is a special case of 01-IP $\implies \mathcal{NP}$.
• $\mathcal{NP}$-completeness can be proved by polynomial transformation from SAT.
• SAT instance: $N = \{1 \ldots n\}$, $C_i = (C_i^+, C_i^-)$ $i = 1 \ldots m$
• set $k = m$
• construct $G = (V, E)$ as follows:
  - $V_i^+ = \{(j, i) : j \in C_i^+\}$
\[ V_i^- = \{(n + j, i) : j \in C_i^-\} \]
\[ V_i = V_i^+ \cup V_i^- \]
\[ V = \bigcup_{i=1}^m V_i \]

- Each pair of nodes in \( V_i \) is joined by an edge; and for \( j = 1 \ldots n \) and \( l \neq i \), nodes \((j, i)\) and \((n + j, l)\) are joined by an edge.
- Let \( x_j = 1 \) if \( j \in N^0 \), \( x_j = 0 \) otherwise.

In general, any stable set of size \( m \) is of the form \( \{(\alpha_1, 1), (\alpha_2, 2) \ldots (\alpha_m, m)\} \) and such a stable set exists if and only if \( N^0 = \{\alpha_i : \alpha_i \leq n\} \) is a solution to the SAT problem.

**Other problems we just proved to be \( \mathcal{NP} \)-complete**

- Given \( G = (V, E) \), the following statements are equivalent:
  1. \( I \) is a stable set of \( G \)
  2. \( V \setminus I \) is a vertex cover for \( G \)
  3. \( I \) is a clique in \( \bar{G} \)

- (lower bound feasibility of) \textit{Maximum Clique} is \( \mathcal{NP} \)-complete.

- A \textit{clique} \( K \) in a graph \( G = (V, E) \) is a subset of vertices \( K \subseteq V \), such that for every two vertices there exists an edge connecting the two.

- Note that a stable set is such iff it is a clique in the complement graph.

  - Given graph \( G = (V, E), k \in \mathbb{Z}_+ \), does \( \exists \ U \subseteq V \) s.t. \( |U| \geq k \) and \( U \) is a clique?
  - Proof: Max Stable Set \( \preceq \) Maximum Clique
  - Given MSS instance: \( G_1 = (V, E), q \). Construct MC instance consisting of \( G = \bar{G}_1 \) and \( k = q \). \( U \) is a clique of size \( k \) in \( G \) iff \( U \) is a stable set of size \( q \) in \( G_1 \).

- (upper bound feasibility of) \textit{Minimum Vertex Cover} is \( \mathcal{NP} \)-complete.

- A \textit{vertex cover} \( C \) in a graph \( G = (V, E) \) is a subset of vertices \( C \subseteq V \), such that each edge of the graph is incident to at least one vertex of the set.

- Note that a stable set is such iff its complement is a vertex cover.

  - Given graph \( G = (V, E), k \in \mathbb{Z}_+ \), does \( \exists \ U \subseteq V \) s.t. \( |U| \leq k \) and \( U \) is a vertex cover?
  - Proof: Max Stable Set \( \preceq \) Minimum Vertex Cover
  - Given MSS instance: \( G_1 = (V, E), q \). Construct MVC instance consisting of \( G = G_1 \) and \( k = |V| - q \). \( U \) is a vertex cover of size \( k \) in \( G \) iff \( V \setminus U \) is a stable set of size \( q \) in \( G_1 \).
Other \textit{NP-complete} problems

• Here we give (without proof) two more \textit{NP-complete} problems.

• Both are decision problems.

• \textit{Partition Problem}: deciding whether a given set \( S \) of positive integers \( a_1 \ldots a_n \) can be partitioned into two subsets \( S_1 \) and \( S_2 \) such that the sum of the numbers in \( S_1 \) equals the sum of the numbers in \( S_2 \) is \textit{NP-complete}.

• \textit{Hamiltonian Circuit (undirected)}: given an undirected graph \( G = (V, E) \), the problem of deciding whether the graph is \textit{Hamiltonian} is \textit{NP-complete}.

• \textit{Hamiltonian Circuit (directed)}: given a di-graph \( G = (V, A) \), the problem of deciding whether the graph is \textit{Hamiltonian}, is \textit{NP-complete}.

• Note that all \textit{complete} graphs are Hamiltonian!