$\mathcal{NP}$-Hard and $\mathcal{NP}$-Complete Problems

Basic concepts

- Solvability of algorithms
  - There are algorithms for which there is no known solution, for example, Turing's Halting Problem
    * Decision problem
    * Given an arbitrary deterministic algorithm A and a finite input I
    * Will A with input I ever terminate, or enter an infinite loop?
    * Alan Turing proved that a general algorithm to solve the halting problem for *all* possible program-input pairs cannot exist
  - Halting problem cannot be solved by any computer, no matter how much time is provided
    * In algorithmic terms, there is no algorithm of any complexity to solve this problem
- Efficient algorithms
  - Efficiency measured in terms of speed
  - For some problems, there is no known efficient solution
  - Distinction between problems that can be solved in polynomial time and problems for which no polynomial time algorithm is known
- Problems classified to belong to one of the two groups
  1. Problems with solution times bound by a polynomial of a small degree
     - Most searching and sorting algorithms
     - Also called tractable algorithms
     - For example, ordered search ($O(\lg n)$), polynomial evaluation ($O(n)$), sorting ($O(n \log n)$)
  2. Problems with best known algorithms not bound by a polynomial
     - Hard, or intractable, problems
     - Traveling salesperson ($O(n^2 2^n)$), knapsack ($O(2^{n/2})$)
     - None of the problems in this group has been solved by any polynomial time algorithm
     - $\mathcal{NP}$-complete problems
       * No efficient algorithm for an $\mathcal{NP}$-complete problem has ever been found; but nobody has been able to prove that such as algorithm does not exist
     - $\mathcal{P} \neq \mathcal{NP}$
       * Famous open problem in Computer Science since 1971
- Theory of $\mathcal{NP}$-completeness
  - Show that many of the problems with no polynomial time algorithms are computationally related
  - The group of problems is further subdivided into two classes

    $\mathcal{NP}$-complete. A problem that is $\mathcal{NP}$-complete can be solved in polynomial time iff all other $\mathcal{NP}$-complete problems can also be solved in polynomial time

    $\mathcal{NP}$-hard. If an $\mathcal{NP}$-hard problem can be solved in polynomial time then all $\mathcal{NP}$-complete problems can also be solved in polynomial time
  - All $\mathcal{NP}$-complete problems are $\mathcal{NP}$-hard but some $\mathcal{NP}$-hard problems are known not to be $\mathcal{NP}$-complete

$$\mathcal{NP}\text{-complete} \subset \mathcal{NP}\text{-hard}$$

- $\mathcal{P}$ vs $\mathcal{NP}$ problems
  - The problems in class $\mathcal{P}$ can be solved in $O(N^k)$ time, for some constant k (polynomial time)

- The problems in class $\mathcal{NP}$ can be *verified* in polynomial time
  - If we are given a *certificate* of a solution, we can verify that the certificate is correct in polynomial time in the size of input to the problem
- Some polynomial-time solvable problems look very similar to $\mathcal{NP}$-complete problems
- Shortest vs longest simple path between vertices
  - Shortest path from a single source in a directed graph $G = (V, E)$ can be found in $O(VE)$ time
  - Finding the longest path between two vertices is $\mathcal{NP}$-complete, even if the weight of each edge is 1
- Euler tour vs Hamiltonian cycle
  - Euler tour of a connected directed graph $G = (V, E)$ is a cycle that traverses each *edge* of G exactly once, although it may visit a vertex more than once; it can be determined in $O(E)$ time
  - A Hamiltonian cycle of a directed graph $G = (V, E)$ is a simple cycle that contains each *vertex* in V
    - Determining whether a directed graph has a Hamiltonian cycle is $\mathcal{NP}$-complete
    - The solution is given by the sequence $\langle v_1, v_2, \ldots, v_{|V|}\rangle$ such that for each $1 \le i < |V|$, $(v_i, v_{i+1}) \in E$
    - The certificate would be the above sequence of vertices
    - It is easy to check in polynomial time that the edges formed by the above sequence are in E, and so is the edge $v_{|V|}, v_1$.
- 2-CNF satisfiability vs. 3-CNF satisfiability
  - Boolean formula has variables that can take value `true` or `false`
  - The variables are connected by operators $\wedge$, $\vee$, and $\neg$
  - A Boolean formula is *satisfiable* if there exists some assignment of values to its variables that cause it to evaluate it to `true`
  - A Boolean formula is in *k-conjunctive normal form* (k-CNF) if it is the AND of clauses of ORs of exactly k variables or their negations
  - 2-CNF: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_2 \vee \neg x_3)$
    - Satisfied by x_1 = `true`, x_2 = `false`, x_3 = `true`
  - We can determine in polynomial time whether a 2-CNF formula is satisfiable but satisfiability of a 3-CNF formula is $\mathcal{NP}$-complete
- $\mathcal{P} \subseteq \mathcal{NP}$
  - Any problem in $\mathcal{P}$ can be solved in polynomial time even without the certificate
  - The open question is whether or not $\mathcal{P} \subset \mathcal{NP}$

- Showing problems to be $\mathcal{NP}$-complete
  - A problem is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and is as "hard" as any problem in $\mathcal{NP}$
  - If *any* $\mathcal{NP}$-complete problem can be solved in polynomial time, then *every* $\mathcal{NP}$-complete problem has a polynomial time algorithm
  - Analyze an algorithm to show how hard it is (instead of how easy it is)
  - Show that no efficient algorithm is likely to exist for the problem
    - As a designer, if you can show a problem to be $\mathcal{NP}$-complete, you provide the proof for its intractability
    - You can spend your time to develop an approximation algorithm rather than searching for a fast algorithm that can solve the problem exactly
  - Proof in terms of $\Omega(n)$

- Decision problems vs optimization problems

  Definition 1 *Any problem for which the answer is either zero or one is called a **decision problem**. An algorithm for a decision problem is termed a **decision algorithm**.*

  Definition 2 *Any problem that involves the identification of an optimal (either minimum or maximum) value of a given cost function is known as an **optimization problem**. An **optimization algorithm** is used to solve an optimization problem.*

- Optimization problems
  - Each feasible solution has an associated value; the goal is to find a feasible solution with the best value
  - SHORTEST PATH problem
    - Given an undirected graph G and vertics u and v
    - Find a path from u to v that uses the fewest edges
    - Single-pair shortest-path problem in an undirected, unweighted graph
- Decision problems
  - The problem gives an answer as "yes" or "no"
  - Decision problem is assumed to be easier (or no harder) to solve compared to the optimization problem
  - Decision problem can be solved in polynomial time if and only if the corresponding optimization problem can
    - If the decision problem cannot be solved in polynomial time, the optimization problem cannot be solved in polynomial time either
- $\mathcal{NP}$-complete problems confined to the realm of decision problems
  - Cast an optimization problem as a related decision problem by imposing a bound on the value to be optimized
  - PATH problem as related to SHORTEST PATH problem
    - Given a directed graph G, vertices u and v, and an integer k, is there a path from u to v with at most k edges?
  - Relationship between an optimization problem and its related decision problem
    - Try to show that the optimization problem is "hard"
    - Or that the decision problem is "easier" or "no harder"
    - We can solve PATH by solving SHORTEST PATH and then comparing the number of edges to k
    - If an optimization problem is easy, its decision problem is easy as well
    - In NP-completeness, if we can provide evidence that a decision problem is hard, we can also provide evidence that its related optimization problem is hard
- Reductions
  - Showing that one problem is no harder or no easier than another also applicable when both problems are decision problems
  - $\mathcal{NP}$-completeness proof – general steps
    - Consider a decision problem A; we'll like to solve it in polynomial time
    - Instance: input to a particular problem; for example, in PATH, an instance is a particular graph G, two particular variables u and v in G, and a particular integer k
    - Suppose that we know how to solve a different decision problem B in polynomial time
    - Suppose that we have a procedure that transforms any instance α of A into some instance β of B with following characteristics:

      Transformation take polynomial time

      Both answers are the same; the answer for α is a "yes" iff the answer for β is a "yes"
  - The above procedure is called a polynomial time *reduction algorithm* and provides us a way to solve problem A in polynomial time
    1. Given an instance α of A, use a polynomial-time reduction algorithm to transform it to an instance β of B
    2. Run polynomial-time decision algorithm for B on instance β
    3. Use the answer for β as the answer for α
  - Using polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B
    - Suppose we have a decision problem A for which we already know that no polynomial-time algorithm can exist
    - Suppose that we have a polynomialtime reduction transforming instances of A to instances of B
    - Simple proof that no polynomial-time algorithm can exist for B

- Nondeterministic algorithms
  - *Deterministic algorithms*
    * Algorithms with uniquely defined results
    * Predictable in terms of output for a certain input
  - Nondeterministic algorithms are allowed to contain operations whose outcomes are limited to a given set of possibilities instead of being uniquely defined
  - Specified with the help of three new $O(1)$ functions
    1. `choice ( S )`
       * *Arbitrarily* chooses one of the elements of set S
       * `x = choice(1,n)` can result in x being assigned any of the integers in the range $[1, n]$, in a completely arbitrary manner
       * No rule to specify how this choice is to be made
    2. `failure()`
       * Signals unsuccessful completion of a computation
       * Cannot be used as a return value
    3. `success()`
       * Signals successful completion of a computation
       * Cannot be used as a return value
       * If there is a set of choices that leads to a successful completion, then one choice from this set must be made
  - A nondeterministic algorithm terminates unsuccessfully iff there exist no set of choices leading to a success signal
  - A machine capable of executing a nondeterministic algorithm as above is called a *nondeterministic machine*
  - Nondeterministic search of x in an unordered array A with $n \geq 1$ elements
    * Determine an index j such that $A[j] = x$ or $j = -1$ if $x \notin A$

```
algorithm nd_search ( A, n, x )
{
// Non-deterministic search
// Input:  A: Array to be searched
// Input:  n: Number of elements in A
// Input:  x: Item to be searched for
// Output: Returns -1 if item does not exist, index of item otherwise

    int j = choice ( 0, n-1 );
    if ( A[j] == x )
    {
        cout << j;
        success();
    }
    cout << -1;
    failure();
}
```

    * By the definition of nondeterministic algorithm, the output is -1 iff there is no j such that $A[j] = x$
    * Since A is not ordered, every deterministic search algorithm is of complexity $\Omega(n)$, whereas the nondeterministic algorithm has the complexity as $O(1)$
  - Nondeterministic sorting algorithm

```
// Sort n positive integers in nondecreasing order

algorithm nd_sort ( A, n )
{
```

```
        // Initialize B[]; B is used for convenience
        // It is initialized to 0 though any value not in A[] will suffice

        for ( i = 0; i < n; B[i++] = 0; );
        for ( i = 0; i < n; i++ )
        {
            j = choice ( 0, n - 1 );

            // Make sure that B[j] has not been used already

            if ( B[j] != 0 ) failure();
            B[j] = A[i];
        }

        // Verify order

        for ( i = 0; i < n-1; i++ )
            if ( B[i] > B[i+1] ) failure();

        write ( B );
        success();
    }
```

- Complexity of `nd_sort` is $\Theta(n)$
  - Best-known deterministic sorting algorithm has a complexity of $\Omega(n \lg n)$
- Deterministic interpretation of nondeterministic algorithm
  - Possible by allowing unbounded parallelism in computation
  - Imagine making n copies of the search instance above, all running in parallel and searching at different index values for x
    - The first copy to reach `success()` terminates all other copies
    - If a copy reaches `failure()`, only that copy is terminated
  - In abstract terms, nondeterministic machine has the capability to recognize the correct solution from a set of allowable choices, without making copies of the program

- Possible to construct nondeterministic algorithms for many different choice sequences leading to successful completions (see `nd_sort`)
  - If the numbers in A are not unique, many different permutations will result into sorted sequence
  - We'll limit ourselves to problems that result in a unique output, or decision algorithms
    - A decision algorithm will output 0 or 1
    - Implicit in the signals `success()` and `failure()`
  - Output from a decision algorithm is uniquely defined by input parameters and algorithm specification
- An optimization problem may have many feasible solutions
  - The problem is to find out the feasible solution with the *best* associated value
  - $\mathcal{NP}$-completeness applies directly not to optimization problems but to decision problems
- Example: Maximal clique
  - Clique is a maximal complete subgraph of a graph $G = (V, E)$
  - Size of a clique is the number of vertices in it
  - Maximal clique problem is an optimization problem that has to determine the size of a largest clique in G

  - Corresponding decision problem is to determine whether G has a clique of size at least k for some given k
  - Let us denote the deterministic decision algorithm for the clique decision problem as `dclique(G, k)`
  - If $|V| = n$, the size of a maximal clique can be found by

    ```
    for ( k = n; dclique ( G, k ) != 1; k-- );
    ```

  - If time complexity of `dclique` is $f(n)$, size of maximal clique can be found in time $g(n) \leq nf(n)$
    * Decision problem can be solved in time $g(n)$
  - Maximal clique problem can be solved in polynomial time iff the clique decision problem can be solved in polynomial time

- Example: 0/1 knapsack
  - Is there a 0/1 assignment of values to x_i, $1 \leq i \leq n$, such that $\sum p_i x_i \geq r$ and $\sum w_i x_i \leq m$, for given m and r, and nonnegative p_i and w_i
  - If the knapsack decision problem cannot be solved in deterministic polynomial time, then the optimization problem cannot either

- Comment on uniform parameter n to measure complexity
  - $n \in \mathcal{N}$ is length of input to algorithm, or input size
    * All inputs are assumed to be integers
    * Rational inputs can be specified by pairs of integers
  - n is expressed in binary representation
    * $n = 10_{10}$ is expressed as $n = 1010_2$ with length 4
    * Length of a positive integer k_{10} is given by $\lfloor \log_2 k \rfloor + 1$ bits
    * Length of 0_2 is 1
    * Length of the input to an algorithm is the sum of lengths of the individual numbers being input
    * Length of input in radix r for k_{10} is given by $\lfloor \log_r k \rfloor + 1$
    * Length of 100_{10} is $\log_{10} 100 + 1 = 3$
    * Finding length of any input using radix $r > 1$
      · $\log_r k = \log_2 k / \log_2 r$
      · Length is given by $c(r)n$ where n is the length using binary representation and $c(r)$ is a number fixed for r
  - Input in radix 1 is in *unary* form
    * $5_{10} = 11111_1$
    * Length of a positive integer k is k
    * Length of a unary input is exponentially related to the length of the corresponding r-ary input for radix r, $r > 1$

- Maximal clique, again
  - Input can be provided as a sequence of edges and an integer k
  - Each edge in $E(G)$ is a pair of vertices, represented by numbers (i, j)
  - Size of input for each edge (i, j) in binary representation is $\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2$
  - Input size of any instance is

    $$n = \sum_{\substack{(i,j) \in E(G) \\ i < j}} \left(\lfloor \log_2 i \rfloor + \lfloor \log_2 j \rfloor + 2 \right) + \lfloor \log_2 k \rfloor + 1$$

    k is the number to indicate the clique size
  - If G has only one connected component, then $n \geq |V|$

  - If this decision problem cannot be solved by an algorithm of complexity $p(n)$ for some polynomial $p()$, then it cannot be solved by an algorithm of complexity $p(|V|)$

- 0/1 knapsack
  - Input size q ($q > n$) for knapsack decision problem is

$$q = \sum_{1 \leq i \leq n} (\lfloor \log_2 p_i \rfloor + \lfloor \log_2 w_i \rfloor) + 2n + \lfloor \log_2 m \rfloor + \lfloor \log_2 r \rfloor + 2$$

  - If the input is given in unary notation, then input size $s = \sum p_i + \sum w_i + m + r$
  - Knapsack decision and optimization problems can be solved in time $p(s)$ for some polynomial $p()$ (dynamic programming algorithm)
  - However, there is no known algorithm with complexity $O(p(n))$ for some polynomial $p()$

Definition 3 *The* ***time required by a nondeterministic algorithm*** *performing on any given input is the minimum number of steps needed to reach a successful completion if there exists a sequence of choices leading to such a completion. In case successful completion is not possible, then the time required is $O(1)$. A nondeterministic algorithm is of complexity $O(f(n))$ if for all inputs of size n, $n \geq n_0$, that result in a successful completion, the time required is at most $cf(n)$ for some constants c and n_0.*

  - Above definition assumes that each computation step is of a fixed cost
    * Guaranteed by the finiteness of each word in word-oriented computers
  - If a step is not of fixed cost, it is necessary to consider the cost of individual instructions
    * Addition of two m-bit numbers takes $O(m)$ time
    * Multiplication of two m-bit numbers takes $O(m^2)$ time
  - Consider the deterministic decision algorithm to get sum of subsets

```
algorithm sum_of_subsets ( A, n, m )
{
// Input: A is an array of integers
// Input: n is the size of the array
// Input: m gives the index of maximum bit in the word

    s = 1                           // s is an m+1 bit word
                                    // bit 0 is always 1
    for i = 1 to n
        s |= ( s << A[i] )          // shift s left by A[i] bits

    if bit m in s is 1
        write ( "A subset sums to m" );
    else
        write ( "No subset sums to m" );
}
```

    * Bits are numbered from 0 to m from right to left
    * Bit i will be 0 if and only if no subsets of $A[j]$, $1 \leq j \leq n$ sums to i
    * Bit 0 is always 1 and bits are numbered $0, 1, 2, \ldots, m$ right to left
    * Number of steps for this algorithm is $O(n)$
    * Each step moves $m + 1$ bits of data and would take $O(m)$ time on a conventional computer
    * Assuming one unit of time for each basic operation for a fixed word size, the complexity of deterministic algorithm is $O(nm)$

- Knapsack decision problem

- Non-deterministic polynomial time algorithm for knapsack problem

```
algorithm nd_knapsack ( p, w, n, m, r, x )
{
// Input: p: Array to indicate profit for each item
// Input: w: Array to indicate weight of each item
// Input: n: Number of items
// Input: m: Total capacity of the knapsack
// Input: r: Expected profit from the knapsack
// Output: x: Array to indicate whether corresponding item is carried or not

    W = 0;
    P = 0;
    for ( i = 1; i <= n; i++ )
    {
        x[i] = choice ( 0, 1 );
        W += x[i] * w[i];
        P += x[i] * p[i];
    }

    if ( ( W > m ) || ( P < r ) )
        failure();
    else
        success();
}
```

- The `for` loop selects or discards each of the `n` items
- It also recomputes the total weight and profit coresponding to the selection
- The `if` statement checks to see the feasibility of assignment and whether the profit is above a lower bound `r`
- The time complexity of the algorithm is $O(n)$
- If the input length is q in binary, time complexity is $O(q)$

• Maximal clique
  - Nondeterministic algorithm for clique decision problem
  - Begin by trying to form a set of k distinct vertices
  - Test to see if they form a complete subgraph

• Satisfiability
  - Let $x_1, x_2, \ldots$ denote a set of boolean variables
  - Let $\bar{x_i}$ denote the complement of x_i
  - A variable or its complement is called a *literal*
  - A *formula* in propositional calculus is an expression that is constructed by connecting literals using the operations *and* ($\wedge$) and *or* ($\vee$)
  - Examples of formulas in propositional calculus
    * $(x_1 \wedge x_2) \vee (x_3 \wedge \bar{x_4})$
    * $(x_3 \vee \bar{x_4}) \wedge (x_1 \vee \bar{x_2})$
  - Conjunctive normal form (CNF)
    * A formula is in CNF iff it is represented as $\wedge_{i=1}^{k} c_i$, where c_i are clauses represented as $\vee l_{ij}$; l_{ij} are literals
  - Disjunctive normal form (DNF)
    * A formula is in DNF iff it is represented as $\vee_{i=1}^{k} c_i$, where c_i are clauses represented as $\wedge l_{ij}$

- Satisfiability problem is to determine whether a formula is true for some assignment of truth values to the variables
  - CNF-satisfiability is the satisfiability problem for CNF formulas
- Polynomial time nondeterministic algorithm that terminates successfully iff a given propositional formula $E(x_1, \ldots, x_n)$ is satisfiable
  - Nondeterministically choose one of the 2^n possible assignments of truth values to $(x_1, \ldots, x_n)$
  - Verify that $E(x_1, \ldots, x_n)$ is true for that assignment

```
algorithm eval ( E, n )
{
    // Determine whether the propositional formula E is satisfiable.
    // Variable are x1, x2, ..., xn

    // Choose a truth value assignment

    for ( i = 1; i <= n; i++ )
        x_i = choice ( true, false );

    if ( E ( x1, ..., xn ) )
        success();
    else
        failure();
}
```

  - The nondeterministic time to choose the truth value is $O(n)$
  - The deterministic evaluation of the assignment is also done in $O(n)$ time

- The classes $\mathcal{NP}$-hard and $\mathcal{NP}$-complete
  - Polynomial complexity
    - An algorithm A is of polynomial complexity if there exists a polynomial $p()$ such that the computation time of A is $O(p(n))$ for every input of size n

Definition 4 *$\mathcal{P}$ is the set of all decision problems solvable by deterministic algorithms in polynomial time. $\mathcal{NP}$ is the set of all decision problems solvable by nondeterministic algorithms in polynomial time.*

  - Since deterministic algorithms are a special case of nondeterministic algorithms, $\mathcal{P} \subseteq \mathcal{NP}$
  - An unsolved problem in computer science is: Is $\mathcal{P} = \mathcal{NP}$ or is $\mathcal{P} \neq \mathcal{NP}$?
  - Cook formulated the following question: Is there any single problem in $\mathcal{NP}$ such that if we showed it to be in $\mathcal{P}$, then that would imply that $\mathcal{P} = \mathcal{NP}$? This led to Cook's theorem as follows:

Theorem 1 *Satisfiability is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.*

- Reducibility
  - Show that one problem is no harder or no easier than another, even when both problems are decision problems

Definition 5 *Let A and B be problems. Problem A **reduces** to B (written as $A \propto B$) if and only if there is a way to solve A by a deterministic polynomial time algorithm using a deterministic algorithm that solves B in polynomial time.*

  - If we have a polynomial time algorithm for B, then we can solve A in polynomial time
  - Reducibility is transitive
    - $A \propto B \wedge B \propto C \Rightarrow A \propto C$

Definition 6 *Given two sets A and $B \in \boldsymbol{N}$ and a set of functions $\boldsymbol{F} : \boldsymbol{N} \rightarrow \boldsymbol{N}$, closed under composition, A is called **reducible** to B ($A \propto B$) if and only if*

$$\exists f \in \boldsymbol{F} \mid \forall x \in \boldsymbol{N},\ x \in A \Leftrightarrow f(x) \in B$$

- Procedure is called polynomial-time *reduction algorithm* and it provides us with a way to solve problem A in polynomial time
  - Also known as *Turing reduction*
  - Given an instance α of A, use a polynomial-time reduction algorithm to transform it to an instance β of B
  - Run the polynomial-time decision algorithm on instance β of B
  - Use the answer of β as the answer for α
  - Reduction from squaring to multiplication
    - All we know is to add, subtract, and take squares
    - Product of two numbers is computed by

$$2 \times a \times b = (a+b)^2 - a^2 - b^2$$

    - Reduction in the other direction: if we can multiply two numbers, we can square a number
  - Computing $(x+1)^2$ from x^2
    - For efficiency sake, we want to avoid multiplication
  - Turing reductions *compute* the solution to one problem, assuming the other problem is easy to solve
- Polynomial-time many-one reduction
  - Converts instances of a decision problem A into instances of a decision problem B
  - Written as $A \leq_m B$; A is many-one reducible to B
  - If we have an algorithm N which solves instances of B, we can use it to solve instances of A in
    - Time needed for N plus the time needed for reduction
    - Maximum of space needed for N and the space needed for reduction
  - Formally, suppose A and B are formal languages over the alphabets Σ and Γ
    - A many-one reduction from A to B is a total computable function $f : \Sigma^* \rightarrow \Gamma^*$ with the property

$$\omega \in A \;\Leftrightarrow\; f(\omega) \in B,\;\; \forall \omega \in \Sigma^*$$

    - If such an f exists, A is many-one reducible to B
  - A class of languages $\boldsymbol{C}$ is *closed* under many-one reducibility if there exists no reduction from a language in $\boldsymbol{C}$ to a language outside $\boldsymbol{C}$
    - If a class is closed under many-one reducibility, then many-one reduction can be used to show that a problem is in $\boldsymbol{C}$ by reducing a problem in $\boldsymbol{C}$ to it
    - Let $S \subset P(\boldsymbol{N})$ (power set of natural numbers), and $\leq$ be a reduction, then S is called closed under $\leq$ if

$$\forall s \in S \;\; \forall A \in \boldsymbol{N} \quad A \leq S \;\Leftrightarrow\; A \in S$$

    - Most well-studied complexity classes are closed under some type of many-one reducibility, including $\mathcal{P}$ and $\mathcal{NP}$
  - Square to multiplication reduction, again
    - Add the restriction that we can only use square function one time, and only at the end
    - Even if we are allowed to use all the basic arithmetic operations, including multiplication, no reduction exists in general, because we may have to compute an irrational number like $\sqrt{2}$ from rational numbers
    - Going in the other direction, however, we can certainly square a number with just one multiplication, only at the end
    - Using this limited form of reduction, we have shown the unsurprising result that multiplication is harder in general than squaring
  - Many-one reductions map *instances* of one problem to *instances* of another
    - Many-one reduction is weaker than Turing reduction
    - Weaker reductions are more effective at separating problems, but they have less power, making reductions harder to design
- Use polynomial-time reductions in opposite way to show that a problem is $\mathcal{NP}$-complete

    * Use polynomial-time reduction to show that no polynomial-time algorithm can exist for problem B
    * $A \subset \boldsymbol{N}$ is called *hard* for S if
$$\forall s \in S \quad s \leq A$$
      $A \subset \boldsymbol{N}$ is called *complete* for S if A is hard for S and A is in S
    * Proof by contradiction
        · Assume that a known problem A is hard to solve
        · Given a new problem B, similar to A
        · Assume that B is solvable in polynomial time
        · Show that every instance of problem A can be solved in polynomial time by reducing it to problem B
        · Contradiction
  – Cannot assume that there is absolutely no polynomial-time algorithm for A

Definition 7 *A problem A is $\mathcal{NP}$-hard if and only if satisfiability reduces to A (satisfiability $\propto A$). A problem A is $\mathcal{NP}$-complete if and only if A is $\mathcal{NP}$-hard and $A \in \mathcal{NP}$.*

  – There are $\mathcal{NP}$-hard problems that are not $\mathcal{NP}$-complete
  – Only a decision problem can be $\mathcal{NP}$-complete
  – An optimization problem may be $\mathcal{NP}$-hard; cannot be $\mathcal{NP}$-complete
  – If A is a decision problem and B is an optimization problem, it is quite possible that $A \propto B$
    * Knapsack decision problem can be reduced to the knapsack optimization problem
    * Clique decision problem reduces to clique optimization problem
  – There are some $\mathcal{NP}$-hard decision problems that are not $\mathcal{NP}$-complete
  – Example: Halting problem for deterministic algorithms
    * $\mathcal{NP}$-hard decision problem, but not $\mathcal{NP}$-complete
    * Determine for an arbitrary deterministic algorithm A and input I, whether A with input I ever terminates
    * Well known that halting problem is undecidable; there exists no algorithm of any complexity to solve halting problem
        · It clearly cannot be in $\mathcal{NP}$
    * To show that "satisfiability $\propto$ halting problem", construct an algorithm A whose input is a propositional formula X
        · If X has n variables, A tries out all the 2^n possible truth assignments and verifies whether X is satisfiable
        · If X is satisfiable, it stops; otherwise, A enters an infinite loop
        · Hence, A halts on input X iff X is satisfiable
    * If we had a polynomial time algorithm for halting problem, then we could solve the satisfiability problem in polynomial time using A and X as input to the algorithm for halting problem
    * Hence, halting problem is an $\mathcal{NP}$-hard problem that is not in $\mathcal{NP}$

Definition 8 *Two problems A and B are said to be* ***polynomially equivalent*** *if and only if $A \propto B$ and $B \propto A$.*

  – To show that a problem B is $\mathcal{NP}$-hard, it is adequate to show that $A \propto B$, where A is some problem already known to be $\mathcal{NP}$-hard
  – Since $\propto$ is a transitive relation, it follows that if satisfiability $\propto A$ and $A \propto B$, then satisfiability $\propto B$
  – To show that an $\mathcal{NP}$-hard decision problem is $\mathcal{NP}$-complete, we have just to exhibit a polynomial time nondeterministic algorithm for it

Polynomial time

- Problems that can be solved in polynomial time are regarded as tractable problems

1. Consider a problem that is solved in time $O(n^{100})$
   - It is polynomial time but sounds intractable
   - In practice, there are few problems that require such a high degree polynomial
2. For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another
3. The class of polynomial-time solvable problems has nice closure properties
   - Polynomials are closed under addition, multiplication, and composition
   - If the output of one polynomial-time algorithm is fed into the input of another, the composite algorithm is polynomial