CHAPTER 14

AN OVERVIEW OF COMPUTATIONAL COMPLEXITY

Mani Nik Farjam - Amin Mohammadzadeh - Mahdi Mahmoudkhani

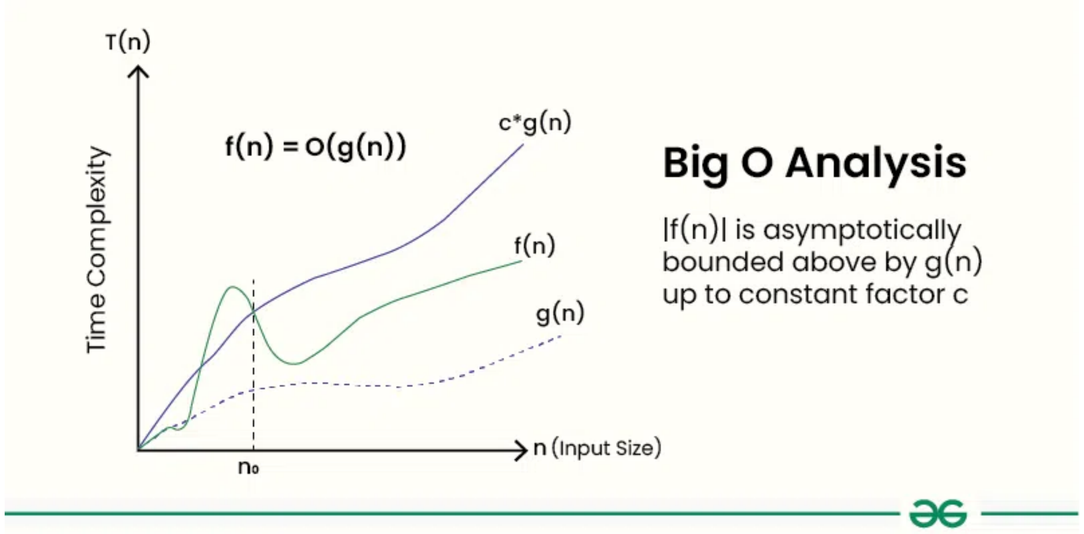
complexity theory: an extensive topic that deals with the efficiency of computation.

- Time Complexity and Space Complexity
- Orders-of-Magnitude expressions

Review:

Big-O Notation

Let ***f(n)*** and ***g(n)*** be functions whose domain is a subset of the positive integers. If there exists a positive constant ***c*** such that for all sufficiently large ***n***, $f(n) \leq c|g(n)|$, we say that ***f* has order at most *g*. We write this as $f(n) = O(g(n))$.**

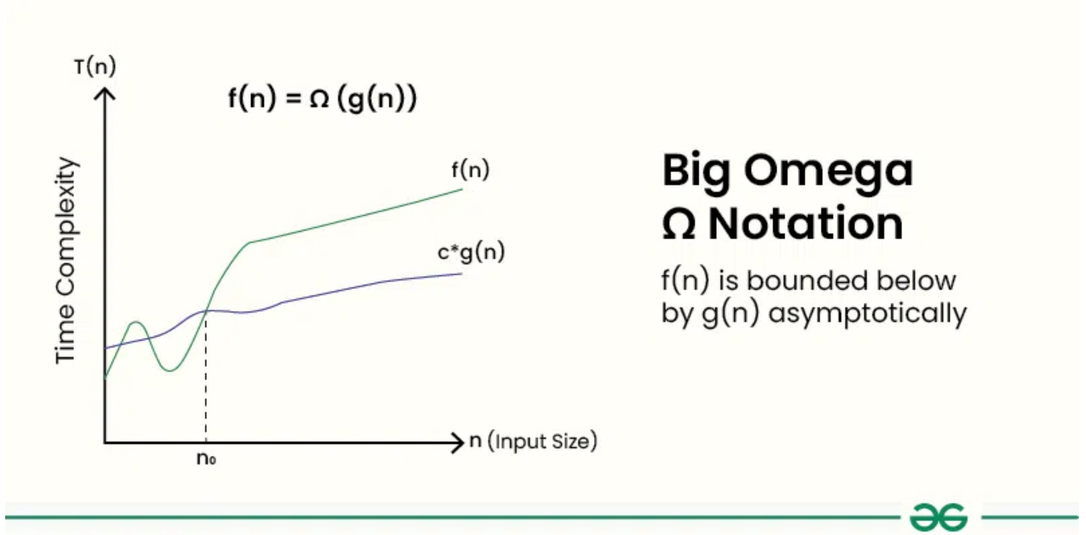


Review:

Big-Omega Notation

Let ***f(n)*** and ***g(n)*** be functions whose domain is a subset of the positive integers. If there exists a positive constant ***c*** such that for all sufficiently large ***n***, ***f(n)*** $\geq c|g(n)|$**, then *f* has order at least *g***.

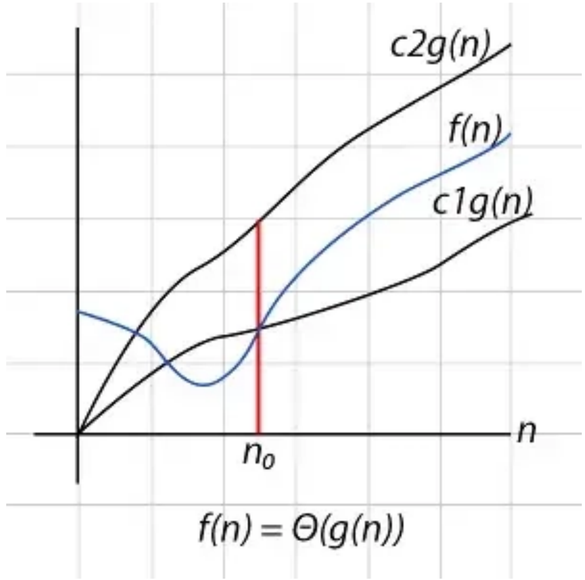
For which we use $f(n) = \Omega(g(n))$**.**



Review:

Theta Notation

Let ***f(n)*** and ***g(n)*** be functions whose domain is a subset of the positive integers. If there exist constants c_1 and c_2 such that **$c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$, *f* and *g* have the same order of magnitude. Expressed as *f(n)* = Θ*(g(n))*.**



14.1

EFFICIENCY OF COMPUTATION

How long does it take to sort a list of 1,000 integers?

Which factors should be taken into account?

- Sorting algorithm used (e.g., quicksort, mergesort, bubblesort)
- Hardware & implementation details

We will make the following simplifying assumptions:

14.1

EFFICIENCY OF COMPUTATION

We will make the following simplifying assumptions:

1. The model for our study will be a Turing machine. The exact type of Turing machine to be used will be discussed below.
2. The size of the problem will be denoted by ***n***. For our sorting problem, ***n*** is obviously the number of items in the list. Although the size of a problem is not always so easily characterized, we can generally relate it in some way to a positive integer.
3. In analyzing an algorithm, we are less interested in its performance on a specific case than in its general behavior. We are particularly concerned with how the algorithm behaves when the problem size increases. Because of this, the primary question involves **how fast the resource requirements grow as n becomes large.**

14.1

EFFICIENCY OF COMPUTATION

1. We give some meaning to the concept of time for a Turing machine. We think of a Turing machine as making one move per time unit, so **the time taken by a computation is the number of moves made.**
2. Here we are interested only in the **worst case** that has the highest resource requirements.

   By saying that a computation has a time-complexity *T(n)*, we mean that the computation for any problem of size *n* can be completed in no more than *T(n)* moves on some Turing machine.

We could analyze algorithms by writing explicit programs and counting the number of steps involved in solving the problem. **But ...**

14.1

EFFICIENCY OF COMPUTATION

1. The number of operations performed may vary with the small details of the program and so may depend strongly on the programmer.
2. From a practical standpoint, we are interested in how the algorithm performs in the **real world**, which may differ considerably from how it does on a Turing machine.

Our attempt at understanding the resource requirements of an algorithm is therefore invariably an **order-of-magnitude** analysis in which we use the O, Ω , and Θ notation.

14.1

Example

Given a set of a numbers $x_1, x_2, \ldots, x_n$ and a key number x, determine if the set contains x.

Unless the set is organized in some way, the simplest algorithm is just a **linear search**.

We know that, in the **worst case**, we have to make n comparisons. We can then say that the time complexity of this linear search is $O(n)$, or even better, $\Theta(n)$.

14.2

TURING MACHINE MODELS AND COMPLEXITY

We have already seen that the efficiency of a computation can be affected by the **number of tapes** of the machine and by whether it is **deterministic or nondeterministic**. As Example 12.8 shows, **nondeterministic solutions are often much more efficient than deterministic alternatives.**

The next example illustrates this even more clearly.

14.2

Example - Satisfiability Problem (SAT)

A logic or boolean constant or variable is one that can take on exactly two values, true or false, which we will denote by **1** and **0**, respectively. Boolean operators are used to combine boolean constants and variables into boolean expressions. The simplest boolean operators are **or**, denoted by $\vee$ and defined by

Variable 1	Variable 2	Or operator
T	T	T
T	F	T
F	T	T
F	F	F

14.2

Example - Satisfiability Problem (SAT)

and the **and** operator ($\wedge$), defined by

Variable 1	Variable 2	And operator
T	T	T
T	F	F
F	T	F
F	F	F

Also needed is **negation**, denoted by a **bar**, and defined by

Variable	~Variable
T	F
F	T

14.2

Example - Satisfiability Problem (SAT)

We consider now boolean expressions in **conjunctive normal form (CNF)**. In this form, we create expressions from variables $x_1, x_2, \ldots, x_n$, starting with $e = t_i \wedge t_j \wedge \ldots \wedge t_k$

The terms $t_i, t_j, \ldots, t_k$ are created by or-ing together variables and their negation, that is,

$t_i = s_l \vee s_m \vee \ldots \vee s_p$

Where each $s_l, s_m, \ldots, s_p$ stands for a variable or the negation of a variable. The s_i will be called **literals**, while the t_i are said to be **clauses** of a CNF expression e.

14.2

Example - Satisfiability Problem (SAT)

The satisfiability problem is then simply stated: Given a satisfiable expression e in conjunctive normal form, find an assignment of values to the variables $x_1, x_2, \ldots, x_n$, that will make the value of e true. For a specific case, look at $\boldsymbol{e_1 = (x_1 \vee x_2) \wedge (x_1 \vee x_3).}$

The assignment $\boldsymbol{x_1 = 0, x_2 = 1, x_3 = 1}$ makes e_1 true so that this expression is satisfiable.

On the other hand, $\boldsymbol{e_2 = (x_1 \vee x_2) \wedge \sim x_1 \wedge \sim x_2}$ is not satisfiable because every assignment for the variables $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$ will make $\boldsymbol{e_2}$ false.

14.2

Example - Satisfiability Problem (SAT)

A deterministic algorithm for the satisfiability problem is easy to discover. We take all possible values for the variables $x_1, x_2, \ldots, x_n$ and for each evaluate the expression. Since there are 2^n such possibilities, this exhaustive approach has **exponential time** complexity.

Again, the nondeterministic alternative simplifies matters. If e is satisfiable, we guess the value of each x_i and then evaluate e. This is essentially an $O(n)$ algorithm.

Exercise: Rewrite the boolean expression $(x_1 \wedge x_2) \vee x_3$ in conjunctive normal form.

14.1

Theorem

Suppose that a two-tape machine can carry out a computation in n steps. Then this computation can be simulated by a standard Turing machine in $O(n^2)$ steps.

Proof: For the simulation of the computation on the two-tape machine, the standard machine keeps the instantaneous description of the two-tape machine on its tape. To simulate one move, the standard machine needs to search the entire active area of its tape. But since one move of the two-tape machine can extend the active area by at most two cells, after n moves the active area has a length of at most $O(n)$. Therefore the entire simulation can be done in $O(n^2)$ moves.

14.2

Theorem

Suppose that a nondeterministic Turing machine *M* can carry out a computation in *n* steps. Then a standard Turing machine can carry out the same computation in $O(k^{an})$ steps, where *k* and *a* are independent of *n*.

Proof: A standard Turing machine can simulate a nondeterministic machine by keeping track of all possible configurations, continually searching and updating the entire active area. If k is the maximum branching factor for the nondeterminism, then after n steps there are at most k^n possible configurations. Since at most one symbol can be added to each configuration by a single move, the length of one configuration after n moves is $O(n)$. Therefore, to simulate one move, the standard machine must search an active area of length $O(nk^n)$, leading to the desired result.

14.3

LANGUAGE FAMILIES AND COMPLEXITY CLASSES

In the **Chomsky hierarchy** for language classification, we associate language families with classes of automata, where each class of automata is defined by the **nature of its temporary storage**.

Another possibility for classifying languages is to use a **Turing machine** and consider **time complexity** a distinguishing factor.

14.1 , 14.2

DEFINITION

We say that a Turing machine *M* decides a language *L* in time T(n) if every *w* in *L* with $|w| = n$ is decided in *T(n)* moves. If M is nondeterministic, this implies that for every $w \in L$, there is at least one sequence of moves of length less than or equal to $T(|w|)$ that leads to acceptance, and that the Turing machine halts on all inputs in time $T(|w|)$.

A language *L* is said to be a member of the class ***DTIME(T(n))*** if there exists a deterministic multitape Turing machine that decides *L* in time $O(T(n))$.

A language *L* is said to be a member of the class ***NTIME(T(n))*** if there exists a nondeterministic multitape Turing machine that decides *L* in time $O(T(n))$.

14.1 , 14.2

DEFINITION

Some relations between these complexity classes, such as

$$DTIME\ (T(n)) \subseteq NTIME(T(n)),$$

and

$$T_1(n) = O(T_2(n))$$

implies

$$DTIME(T_1(n)) \subseteq DTIME\ (T_2(n)),$$

14.3

THEOREM

For every integer $k \geq 1$,

$$DTIME(n^k) \subset DTIME\ (n^{k+1})$$

Proof: This follows from a result in Hopcroft and Ullman (1979, p.299).

Conclusion: There are some languages that can be decided in time $O(n^2)$ for which there is no linear-time membership algorithm, that there are languages in $DTIME(n^3)$ that are not in $DTIME(n^2)$, and so on. This gives us an infinite number of nested complexity classes. We get even more if we allow exponential time complexity. In fact, there is no limit to this; no matter how rapidly the complexity function $T(n)$ grows, there is always something outside $DTIME(T(n))$.

14.4

THEOREM

There is no total Turing computable function $f(n)$ such that every recursive language can be decided in time $f(n)$, where n is the length of the input string.

Proof: Consider the alphabet $\Sigma = \{0, 1\}$, with all strings in Σ^+ arranged in proper order $w_1, w_2, \ldots$ Also, assume that we have a proper ordering for the Turing machines in $M_1, M_2, \ldots$

Assume now that the function $f(n)$ in the statement of the theorem exists. We can then define the language $L = \{w_i : M_i \text{ does not decide } w_i \text{ in } f(|w_i|) \text{ steps}\}$. **(14.4)**

14.4

THEOREM

We claim that L is recursive. To see this, consider any $w \in L$ and compute first $f(|w|)$. By assuming that f is a total Turing computable function, this is possible. We next find the position i of w in the sequence $w_1, w_2, \ldots$ This is also possible because the sequence is in proper order. When we have i, we find M_i and let it operate on w for $f(|w|)$ steps. This will tell us whether or not w is in L , so is recursive.

14.4

THEOREM

But we can now show that L is not decidable in time $f(n)$. Suppose it were. Since L is recursive, there is some M_k, such that $L = L(M_k)$. Is w_k in L? If we claim that if w_k is in L, then M_k decides w y in $f(|w_k|)$ steps. But this contradicts **(14.4)**. Conversely, we get a contradiction if we assume that $w_k \notin L$. The inability to resolve this issue is a typical diagonalization result and leads us to conclude that the original assumption, namely the existence of a computable $f(n)$, must be false.

Theorem 14.3 allows us to make some claims, for example, that there is a language in $DTIME(n^4)$ that is not in $DTIME(n^2)$. Although this may be of theoretical interest, it is not clear that such a result has any practical significance. At this point, we have no clue what the characteristics of a language in $DTIME(n^4)$ might be. We can get a little more insight into the matter if we relate the complexity classification to the languages in the **Chomsky hierarchy**. We will look at some simple examples that give some of the more obvious results.

14.3

Example

Every regular language can be recognized by a deterministic finite automaton in time proportional to the length of the input. Therefore,

$$L_{REG} \subseteq DTIME(n)$$

But $DTIME(n)$ includes much more than L_{REG}. We have already established in Example-13.7 that the context-free language $\{a^n b^n : n \geq 0\}$ can be recognized by a two-tape machine in time $O(n)$. The argument given there can be used for even more complicated languages.

14.4

Example

The non-context-free language $L = \{ww: w \in \{a, b\}^*\}$ is in $NTIME(n)$. This is straightforward, as we can recognize strings in this language by the following algorithm:

1. Copy the input from the input file to tape 1. Nondeterministically guess the middle of this string.
2. Copy the second part to tape 2.
3. Compare the symbols on tape 1 and tape 2 one by one.

Clearly, all of the steps can b e done in $O(|w|)$ time, so $L \in NTIME(n)$.

14.4

Example

Actually, we can show that $L \in DTIME(n)$ if we can devise an algorithm for finding the middle of a string in $O(n)$ time. This can be done: We look at each symbol on tape 1, keeping a count on tape 2 , but counting only every second symbol.

14.5

Example

It follows from Example 12.8 that $L_{CF} \subseteq DTIME(n^3)$ and $L_{CF} \subseteq NTIME(n)$.

Consider now the family of context-sensitive languages. Exhaustive search parsing is possible here also since only a limited number of productions are applicable at each step. Following the analysis leading to **Equation (5.2)**, we see that the maximum number of sentential forms is $N = |P| + |P|^2 + \ldots + |P|^{cn} = O(|P|^{cn+1})$.

Note, however, that we cannot claim from this that $L_{CS} \subseteq DTIME(|P|^{cn+1})$.
because we cannot put an upper bound on $|P|$ and c.

Exercise: Show that $L = \{www: w \in \{a, b\}^+\}$ is in $DTIME(n)$.

14.4

P and NP Complexity Classes

At this point, we have encountered some difficulties in trying to find useful complexity classes for languages:

1. There are infinite nested complexity classes $DTIME(n^k), k=1,2,\ldots$ and they barely have any connection to the Chomsky hierarchy.
2. The particular model of Turing machine, even if we restrict ourselves to deterministic machines, affects the complexity.
3. We have found several languages that can be decided efficiently by a nondeterministic Turing machine. For some, there are also reasonable deterministic algorithms, but for others we know only inefficient, brute-force methods. What is the implication of these examples?

14.4

P and NP Complexity Classes

Let us ignore some factors that are less important, for example by removing the distinction between ***DTIME(n^k)*** and ***DTIME(n^{k+1})***. We can argue that the difference between, say, ***DTIME(n)*** and ***DTIME(n^2)*** is not fundamental, since some of it depends on the specific TM we have (e.g. how many tapes).
This leads us to consider the famous complexity class

$$P = \bigcup_{i \geq 1} DTIME\left(n^i\right)$$

This class includes all languages that are accepted by some deterministic Turing machine in polynomial time, without any regard to the degree of the polynomial.

14.4

P and NP Complexity Classes

Since the distinction between deterministic and nondeterministic complexity classes appears to be fundamental, we also introduce

$$NP = \bigcup_{i \geq 1} NTIME\left(n^i\right)$$

It is obvious that $P \subseteq NP$.

While it is generally believed that there are some languages in ***NP*** that are not in ***P***, no one has yet found an example of this.

Exercise: Show that P is closed under union, intersection, and complementation.

14.4

Intractable problems

The interest in these complexity classes, particularly in the class ***P***, comes from an attempt to distinguish between realistic and unrealistic computations. Certain computations, although theoretically possible, have such high resource requirements that in practice they must be rejected as unrealistic on existing computers, as well as on supercomputers yet to be designed. Such problems are sometimes called intractable to indicate that, while in principle computable, there is no realistic hope of a practical algorithm.

But how do we define intractability?

14.4

Cook-Karp Thesis

In the **Cook-Karp** thesis, a problem that is in ***P*** is called tractable, and one that is not is said to be intractable.
But is the Cook-Karp thesis a good way of separating problems we can solve realistically from those we cannot?

$$DTIME(2^{0.1n})$$

$$DTIME(n^{100})$$

The justification for the Cook-Karp thesis seems to lie in the empirical observation that most practical problems in ***P*** are in ***DTIME (n)***, ***DTIME (n^2)***, *or* ***DTIME (n^3)***, while those outside this class tend to have exponential complexities.

14.5

Some NP Problems - SAT

We can show that SAT is in ***NP***.

Suppose that a CNF expression has length ***n***, with m different literals. Since clearly $m < n$, we can take *n* as the problem size.

Next, we must encode the CNF expression as a string for a Turing machine: $\Sigma = \{x, \vee, \wedge, (,), -, 0, 1\}$.

We will encode the subscript of ***x*** as a binary number (for example x_3 will be encoded to x_{11}).

Since the subscript cannot be larger than m, the maximum encoded length of any subscript is ***logm***. As a consequence, the maximum length of encoded CNF is in $O(n \log n)$.

Nondeterministically generate trial solution in $O(n)$.

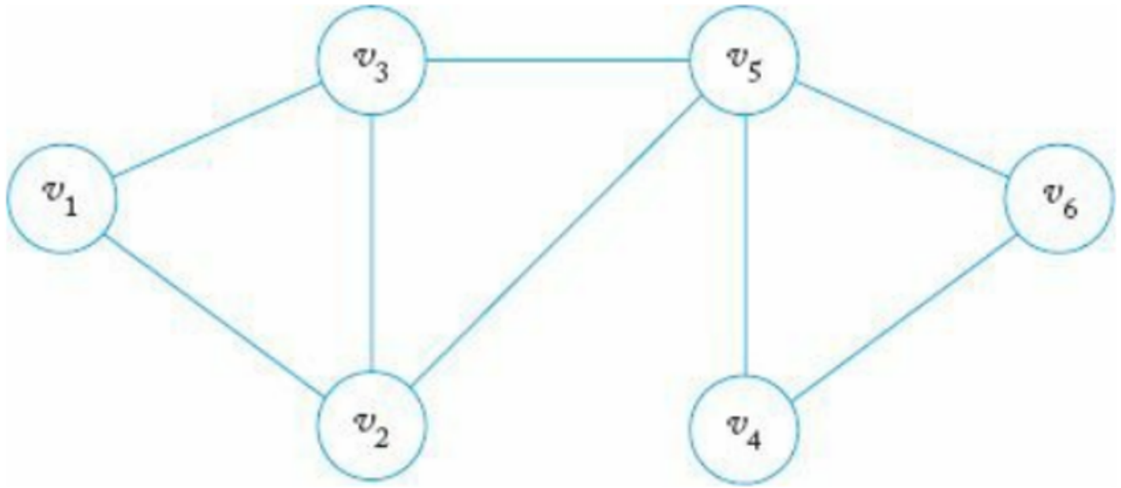
Then we need to substitute the trial solution in the input string, which can be done in $O(n^2 \log n)$.

So, the entire processes can be done in $O(n^3)$, which means $SAT \in NP$.

14.5

Some NP Problems - HAMPATH

Hamiltonian path problem: Given a graph G, decide if it has a Hamiltonian path.



14.5

Some NP Problems - HAMPATH

We can show that **HAMPATH** is in ***NP***. Assume the input (graph representation) is an $n \times n$ adjacency matrix. So the input string will have a size of $O(n^2)$.

1. Create a list of numbers $1,2,\ldots,n$ in unary notation. Since the length of this list is $O(n^2)$ (why?), this can be done in $O(n^2)$.
2. Scan the list and nondeterministically select one number. If a number is selected, add it to the permutation and remove it from the list.
3. Repeat step2 n times.

This will take $O(n^3)$ time. It's trivial that we can use the adjacency matrix to check if the permutation forms a path in polynomial time. Therefore **HAMPATH** $\in$ ***NP***.

14.5

Some NP Problems - CLIQ

Let G be an undirected graph with vertices $v_1, \ldots, v_n$. A *k-clique* is a subset $V_k \subseteq 2^V$, such that there is an edge between every pair of vertices $v_i, v_j \in V_k$. The clique problem (*CLIQ*) is to decide if, for a given k, G has a k-clique.

Exercise: Show that ***CLIQ*** is in ***NP***.

14.6

Polynomial-Time Reduction

A language L_1 is said to be **polynomial-time reducible** to another language L_2 if there exists a deterministic Turing machine by which any w_1 in the alphabet of L_1 can be transformed in polynomial time to a w_2 in the alphabet of L_2 in such a way that $w_1 \in L_1$ if and only if $w_2 \in L_2$.

14.6

Polynomial-Time Reduction - 3SAT

A restricted type of satisfiability is the three-satisfiability problem (3-SAT) in which each clause can have at most three literals. We want to show that SAT problem is polynomial-time reducible to 3-SAT.

We will process each clause in the CNF formula separately:

Case 1: If the clause has exactly 3 literals, leave it unchanged.

Case 2: If the clause has 1 literal (x), replace it with $(x \vee x \vee x)$.

Case 3: If the clause has 2 literals $(x \vee y)$, replace it with $(x \vee y \vee y)$.

Case 4: Clause has more than 3 literals:

Let $C = (x_1 \vee x_2 \vee \ \ldots \vee x_k)$, $k > 3$. Introduce new variables $y_1, y_2, \ldots, y_{k-3}$. Replace the clause with:

$$(x_1 \vee x_2 \vee y_1)\ (\neg y_1 \vee x_3 \vee y_2)\ (\neg y_2 \vee x_4 \vee y_3) \ldots (\neg y_{k-4} \vee x_{k-2} \vee y_{k-3})\ (\neg y_{k-3} \vee x_{k-1} \vee x_k)$$

14.6

Polynomial-Time Reduction - 3SAT

Correctness:

(→) If φ is satisfiable, then $f(\varphi)$ is satisfiable.

(←) If $f(\varphi)$ is satisfiable, then φ is satisfiable.

We also need to show that this transformation can be done in polynomial time.

Let's prove these on the whiteboard.

Exercise: Show that 3-SAT is polynomial-time reducible to CLIQ.

14.7

NP-Completeness

A language L is said to be NP complete if $L \in NP$ and every $L' \in NP$ is polynomial-time reducible to L.

It follows from this definition that if L is NP-complete and polynomial-time reducible to L_1, then L_1 is also **NP-complete**. So, if we can find one deterministic polynomial-time algorithm for any NP-complete language, then every language in NP is also in P, that is, $P = NP$.

On the other hand, if one could prove that any of the many known NP-complete problems is intractable, then

$$P \neq NP \text{ (why?)}$$

This puts NP-completeness in the center of the question of the tractability of many important problems.

14.7

Cook's Theorem

The SAT problem is **NP-complete**.
Proof of the theorem is outside of the scope of the book.
We have shown that SAT can be reduced to 3-SAT, and that 3-SAT can be reduced to CLIQ. Therefore, 3-SAT and CLIQ are both NP-complete (why?).

The End

Thanks for your attention :)