Generalized Nondeterministic Finite Automaton (NFA)

Definition: Let A and B be languages. We define the regular operations

\[
\begin{align*}
\text{Union:} & \quad A \cup B = \{ x \mid x \in A \text{ or } x \in B \} \\
\text{Intersection:} & \quad A \cap B = \{ x \mid x \in A \text{ and } x \in B \} \\
\text{Complement:} & \quad \overline{A} = \{ x \mid x \notin A \} \\
\text{Star:} & \quad A^* = \{ x_1 x_2 \cdots x_n \mid x_i \in A \text{ for } i = 1, 2, \ldots, n \} \\
\text{Concatenation:} & \quad A B = \{ x y \mid x \in A \text{ and } y \in B \}
\end{align*}
\]

Finite Automata

- Nondeterministic finite automata are useful in several respects. As we will show,

\[
\begin{align*}
\text{Nondeterminism is a generalization of determinism, so every deterministic finite}
\end{align*}
\]

- A finite automaton is a list of five objects:

\[
\begin{align*}
Q & \quad \text{is a finite set called the states.} \\
\Sigma & \quad \text{is the alphabet.} \\
\delta & \quad \text{is the transition function, } \delta: Q \times \Sigma \rightarrow Q. \\
F & \quad \text{is the set of accept states.} \\
S & \quad \text{is the start state.}
\end{align*}
\]

The output of an finite automaton is accepted if the automaton is now in an accept state.

- The DFA (deterministic finite automaton) always has exactly one exiting transition arrow for each symbol in the alphabet.

- In a NFA (nondeterministic finite automaton), every state always has exactly one exiting transition arrow for each symbol in the alphabet. In an NFA (nondeterministic finite automaton) a state may have zero, one or many exiting transition arrows.

- Nondeterministic finite automata are useful in several respects. As we will show, you only need to remember certain crucial info.

A language is called a regular language if some finite automaton recognizes it.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.

- Deterministic and nondeterministic finite automaton recognize the same class of languages.

- Two machines are equivalent if they recognize the same language.

- Every NFA has an equivalent DFA.

- If A is the number of states of the NFA, it has 2^A substrates of states. Each substrate corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^A states.
**NON-REGULAR LANGUAGES**

To understand the power of finite automata, you must also understand their limitations. In this section, we show how to prove that certain languages cannot be recognized by any finite automaton.

Let’s take the language $B = \{x | x \text{ has length } p \}$. If we attempt to find a DFA that recognizes $B$, we discover that the machine seems to need to remember how many Os have been seen so far, as it reads the input.

Because the number of Os isn’t limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.

Our technique for proving nonregularity stems from a theorem about regular languages, traditionally called the pumping lemma. This theorem states that all regular languages have a special property. If we can prove that a language does not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be "pumped." If they are at least as long as a certain special value, called the pumping length. That means each new string contains a section that can be repeated any number of times with the resulting string remaining in the language.

**Example 1**

Let $B = \{0^n1^n | n \geq 0 \}$

- $B$ is regular and the pumping length is $p = 2$
- Suppose $s \in B$ and $s = 0^p1^p$.
- By our assumption, there exists $i \geq 1$ such that $xy^iz \in B$.
- $|y| \geq 1$ and $|xy| < p$.
- By the condition 3, $|y| > 0$.
- This $B$ is not regular (contradiction)

**Example 2**

Let $C = \{w \in \{0,1\}^* | w \text{ has an equal number of } 0s \text{ and } 1s \}$

- Suppose $s \in C$ and $s = 0^p1^p$.
- By the condition 3, $|y| \leq p$.
- $xy^iz \in C$.
- Thus $C$ is not regular.

**Example 3**

Let $D = \{a^n b^n | n \geq 0 \}$

- Suppose $s \in D$ and $s = a^p b^p$.
- By the condition 3, $|y| \leq p$.
- $xy^iz \in D$.
- Thus $D$ is not regular.

**Example 4**

Let $E = \{a^n b^n c^n | n \geq 0 \}$

- Suppose $s \in E$ and $s = a^p b^p c^p$.
- By the condition 3, $|y| \leq p$.
- $xy^iz \in E$.
- Thus $E$ is not regular.

**Example 5**

Let $F = \{a^n b^n c^n | n \geq 0 \}$

- Suppose $s \in F$ and $s = a^p b^p c^p$.
- By the condition 3, $|y| \leq p$.
- $xy^iz \in F$.
- Thus $F$ is not regular.

**Example 6**

Let $G = \{a^n b^n c^n | n \geq 0 \}$

- Suppose $s \in G$ and $s = a^p b^p c^p$.
- By the condition 3, $|y| \leq p$.
- $xy^iz \in G$.
- Thus $G$ is not regular.

**Example 7**

Let $H = \{a^n b^n c^n | n \geq 0 \}$

- Suppose $s \in H$ and $s = a^p b^p c^p$.
- By the condition 3, $|y| \leq p$.
- $xy^iz \in H$.
- Thus $H$ is not regular.

**Example 8**

Let $I = \{a^n b^n c^n | n \geq 0 \}$

- Suppose $s \in I$ and $s = a^p b^p c^p$.
- By the condition 3, $|y| \leq p$.
- $xy^iz \in I$.
- Thus $I$ is not regular.
DESIGNING CONTEXT-FREE GRAMMARS

If A is a context-free language, then there is a number \( p \) (the pumping length) where, if \( s \) is a string of \( A \), then \( s = uvwxy \) with \( v \) and \( x \) are nonempty, and \( |vx| \leq p \), and for all \( i \geq 0 \), \( uv^ixy \in A \).

1. We add a new start variable \( S_i \) and the rule \( S_i \rightarrow S \) (guarantees that the start variable occurs only on the left-hand side.)
2. We remove all \( i \) rules (in the form \( A \rightarrow w \)) where \( A \) is \( S_i \).
3. If we have the rule \( A \rightarrow \alpha \beta \), then we add \( A \rightarrow \alpha \gamma \beta \) and \( \gamma \rightarrow \epsilon \).
4. If \( i \) rules have been removed, then \( p \) rules must be added.
5. If \( i > p \), then \( p \) rules can be pumped, but we show that it cannot.

There is a language \( L \) such that \( L \subseteq A \) and \( \epsilon \notin L \) and \( L \) is not regular.

**Example:** \( A \rightarrow B \) \( B \rightarrow B + b \) \( B \rightarrow b \)

1. \( S \rightarrow ASA \mid AB \) (all new start \( S_i \) Remove \( i \) rules \( \{1, 2, 6\} \)
2. Remove rules \( \{1, 2\} \) (all new start \( S_i \) Remove \( i \) rules \( \{1, 2\} \)
3. Remove rules \( \{1, 2\} \) (all new start \( S_i \) Remove \( i \) rules \( \{1, 2\} \)
4. Remove rules \( \{1, 2\} \) (all new start \( S_i \) Remove \( i \) rules \( \{1, 2\} \)
5. Remove rules \( \{1, 2\} \) (all new start \( S_i \) Remove \( i \) rules \( \{1, 2\} \)
6. Remove rules \( \{1, 2\} \) (all new start \( S_i \) Remove \( i \) rules \( \{1, 2\} \)

**PUSHDOWN AUTOMATA (PDA)**

- These automata are like NFA but have an extra component called a stack. The stack provides additional memory beyond the finite control. The stack \( \Sigma \) is the input alphabet, and \( \Gamma \) is \( \Sigma \) extended with a special bottom symbol \( \varepsilon \).
- A PDA is a 6-tuple \((Q, \Gamma, \Sigma, \delta, S, F)\), where \( Q \) is the set of states, \( \Gamma \) is the tape alphabet, \( \Sigma \) is the input alphabet, \( F \) is the set of accept states, and \( \delta \) is a transition function.

**Definition:** A pushdown automaton is a 6-tuple \((Q, \Gamma, \Sigma, \delta, S, F)\) where \( Q \) is the set of states, \( \Gamma \) is the tape alphabet, \( \Sigma \) is the input alphabet, \( F \) is the set of accept states, and \( \delta \) is a transition function.

- **Deterministic pushdown automata (DPDA):** A PDA is deterministic if all possible transitions for a given state and input symbol lead to only one state.
- **Non-deterministic pushdown automata (NPDA):** A PDA is non-deterministic if multiple transitions are possible for a given state and input symbol.

**Example:**

- **NPDA:** \( Q = \{ q_0, q_1 \}, \Gamma = \{ a, b, \cdot \}, \Sigma = \{ \cdot \}, S = q_0, F = \{ q_1 \} \)
- **DPDA:** \( Q = \{ q_0, q_1 \}, \Gamma = \{ a, b, \cdot \}, \Sigma = \{ \cdot \}, S = q_0, F = \{ q_1 \} \)

**EQUIVALENCE WITH CONTEXT-FREE GRAMMARS**

A language is context-free if and only if some pushdown automaton recognizes it.

**Converting a CFG to a PDA method:**

1. Place the marker symbol \( S \) and the start variable on the stack.
2. Repeat the following steps for \( (s, e) \):
   - If the top of the stack is a variable \( A \), non-deterministically select one of the rules for \( A \) and substitute \( \Gamma A \) for the string on the right-hand side.
   - If the top of the stack is a terminal symbol \( a \), read the symbol from the input and compare it to \( a \).
   - If they match, repeat (consume it from the input and the method above resumes the processing of the stack). If they do not match, reject this branch of the non-determinism.
   - If the top of the stack is the marker \( S \), enter the accept state. Doing so accepts the input if it has been read.

**Converting a PDA to a CFG method:**

1. If \( P \) has a single accept state \( q_0 \), then \( P \) is deterministic.
2. If \( P \) has multiple accept states, then \( P \) is non-deterministic.
3. Each transition \( (s, e) \rightarrow (s', e') \) gives us a non-terminal symbol \( a \) in the transition function \( \delta \), with rule \( S \rightarrow a \). The rules \( S \rightarrow a \) are the non-terminal symbols in the grammar.

**PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES**

If \( A \) is a context-free language, then there is a number \( p \) (the pumping length) where, if \( s \) is any string in \( A \) of length at least \( p \), then \( s \) may be divided into five pieces \( s = uvxyz \) satisfying the conditions:

- \( |vxy| \leq p \)
- \( |vy| \leq p \)
- \( uv^nxy \in A \) for all \( i \geq 0 \)

We say that every context-free language \( A \) is \( \text{pumpable} \).

**Example:**

- \( A = \{ a^n b^m c^n d^m \mid n, m \geq 0 \} \)
- \( s = a^2 b^2 c^2 d^2 \)
- \( s = uvxyz \) with \( |vxy| \leq p \)
- \( |vy| \leq p \)
- \( uv^nxy \in A \) for all \( i \geq 0 \)

- \( s = a^2 b^2 c^2 d^2 \)
- \( s = uvxyz \) with \( |vxy| \leq p \)
- \( |vy| \leq p \)
- \( uv^nxy \in A \) for all \( i \geq 0 \)

**Example:**

- \( A = \{ a^n b^n \mid n \geq 0 \} \)
- \( s = a^2 b^2 \)
- \( s = uvxyz \) with \( |vxy| \leq p \)
- \( |vy| \leq p \)
- \( uv^nxy \in A \) for all \( i \geq 0 \)

We say that every context-free language \( A \) is \( \text{pumpable} \).

**Example:**

- \( A = \{ a^n b^m c^n d^m \mid n, m \geq 0 \} \)
- \( s = a^2 b^2 c^2 d^2 \)
- \( s = uvxyz \) with \( |vxy| \leq p \)
- \( |vy| \leq p \)
- \( uv^nxy \in A \) for all \( i \geq 0 \)

We say that every context-free language \( A \) is \( \text{pumpable} \).

- \( s = a^2 b^2 c^2 d^2 \)
- \( s = uvxyz \) with \( |vxy| \leq p \)
- \( |vy| \leq p \)
- \( uv^nxy \in A \) for all \( i \geq 0 \)

We say that every context-free language \( A \) is \( \text{pumpable} \).
TURING MACHINES

- Similar to a finite automaton but with an unlimited and unrestricted memory, a Turing machine is a much more accurate model of a general purpose computer. A Turing machine can do everything that a real computer can do, hence, even a Turing machine cannot solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation.

- The following list summarizes the differences between finite automata and Turing machines:
  1. A Turing machine can both write on the tape and read from it.
  2. The write–read head can move both to the left and to the right.
  3. The tape is infinite.
  4. The special states for rejecting and accepting have immediate effect.

- Actually we almost never give formal descriptions of Turing machines because they tend to be very big.

- Definition: A Turing machine is a 7-tuple \((M, Q, \Delta, q_0, \alpha, \beta, F)\), where \(Q\) is the set of states, \(\Delta\) is the input alphabet not containing the special blank symbol \(\_\), \(q_0\) is the start state, \(\alpha\) and \(\beta\) are special states, and \(F\) is the set of accepting states.

- As a Turing machine computes, changes occur in the current state, the current tape content, and the rest of the tape is blank.

- The following list summarizes the differences between finite automata and Turning machines:
  1. A configuration is represented as \(uqv\) where \(u\) and \(v\) in \(\Gamma^m\) and \(q\) in \(Q\):
     - The current state is \(q\).
     - The current tape content is \(u\).
     - The current head of the tape is \(q\).
     - The rest of the tape is blank.

- Properties of configurations:
  - The start configuration is \(q_0v\).
  - In accepting configuration the state is \(q\).
  - In rejecting configuration the state is \(\beta\).
  - Accepting and rejecting configurations are halting configurations.

EXAMPLES OF TURING MACHINES

- Machine \(M_1\) accepts \(L_1 = \{0^n1^n | n \geq 0\}\):

  Machine \(M_1\) State Diagram:

  ![Machine M1 State Diagram](image)

  - **On input string \(w\):**
    1. Start left to right across the tape, crossing off every other 0.
    2. If in stage 1 the tape contained a single 0, accept.
    3. If in stage 2 the tape contained more than a single 0 and the number of 0s was odd, reject.
    4. Return the head to the left-hand end of the tape.
    5. Go to stage 1.

- Machine \(M_2\) accepts \(L_2 = \{0^n1^n | n \geq 0\}\):

  Machine \(M_2\) State Diagram:

  ![Machine M2 State Diagram](image)

  - **On input string \(w\):**
    1. Scan the input string \(w\) left to right to determine whether it is a member of \(0^n1^n\).
    2. If it is not, reject.
    3. Rewind the tape back to the left-hand end of the tape.
    4. Cross off an \(a\) and scan to the right until a 0 occurs. Shuttle between the 0s and 1s, crossing off one each until all 0s are gone. If all 0s have been crossed off and some 1s remain, reject.
    5. Return the head to the left-hand end of the tape.

- Machine \(M_3\) accepts \(L_3 = \{0^n1^n | n \geq 0\}\):

  Machine \(M_3\) State Diagram:

  ![Machine M3 State Diagram](image)

  - **On input string \(w\):**
    1. If \(w\) is accepted, reject.
    2. Otherwise, accept.

PROPERTIES OF TURING-DECIDABLE LANGUAGES

- The collection of Turing-decidable languages is closed under union, concatenation, star, and intersection (CUT complementation).

- The collection of Turing-recognizable languages is closed under union, concatenation, star, and intersection (CUT complementation).

- The computation of a non-deterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. If you want to simulate a non-deterministic TM with a "normal" TM you have to perform a breadth-first search through the tree, because with depth-first you can lose yourself in an infinite branch of the tree and miss some branches of the computation leading to the accept state. The machine accepts its input.

- Every non-deterministic Turing machine has an equivalent deterministic Turing machine.

- A language is Turing-recognizable if and only if some non-deterministic Turing machine recognizes it.

- We call a non-deterministic Turing machine a decider if all branches halt on all inputs.

- A language is decidable if and only if some deterministic TM recognizes it.

- Loosely defined, an enumerator is a Turing machine with an attached printer.

- A language is Turing-recognizable if and only if some enumerator enumerates it.
UNDECIDABLE PROBLEMS FROM LANGUAGE THEORY

TURING DECIDABLE / TURING RECOGNIZABLE

The Church-Turing thesis: intuitively a notion of algorithm is equal to Turing machine algorithms.

Our notation for the encoding of an object O into its representation as a string is <O>: if we have several objects Q0, Q1, ..., Qk we denote their encoding into a single string by <Q0, Q1, ..., Qk>.

An algorithm always stops.

Chapter 4: Decidability

Acceptance problem expressed as languages for regular expressions:

\[ \text{A}_{\text{DFA}} = \{<A,w>| A \text{ is a DFA that accepts input string } w\} \]

\[ \text{A}_{\text{NFA}} = \{<A,w>| A \text{ is an NFA that accepts input string } w\} \]

Theorem: Consider \( A \text{ a DFA} \) that accepts an input string \( w \).

The problem of testing whether a DFA \( A \) accepts an input \( w \) is the same as the problem of testing whether \( <A,w> \) is a member of the language \( \text{DFA} \).

We can formulate other computational problems in term of testing membership in a language. Showing that the language is decidable is the same as showing that the computational problem is decidable.

Language inclusion testing for regular languages:

\[ \text{INC}_{\text{DFA}} = \{<A,B>| A \text{ and } B \text{ are DFAs and } L(A) \subseteq L(B)\} \]

Language equivalence testing for context-free grammars:

\[ \text{EQ}_{\text{CFG}} = \{<G,w>| G \text{ is a CFG and } w \in L(G)\} \]

Language decidability testing for regular languages:

\[ \text{DEC}_{\text{DFA}} = \{<A>| A \text{ is a DFA and } L(A) \neq \emptyset\} \]

Language inclusion testing for context-free languages:

\[ \text{INC}_{\text{CFG}} = \{<G,H>| G \text{ and } H \text{ are CFGs and } L(G) \subseteq L(H)\} \]

Language equivalence testing for context-free grammars:

\[ \text{EQ}_{\text{CFG}} = \{<G,H>| G \text{ and } H \text{ are CFGs and } L(G) = L(H)\} \]

Turing decidable / Turing recognizable

Some languages are not Turing recognizable, for the reason that there are uncountable many languages for only countably many Turing machines. Because each Turing machine may accept a single language and there are more languages than Turing machines, some languages are not recognizable by any Turing machine.

Chapter 5: Undecidability

Undecidable problems from language theory:

- Some languages are not Turing recognizable.
- The following theorem shows that, if both a language and its complement are Turing recognizable, the language is decidable. Hence, for any undecidable language, either it or its complement is not Turing recognizable. We say that a language to be Turing recognizable if it is the complement of a Turing recognizable language.
- A language is decidable if it is both Turing recognizable and co-Turing recognizable.
- A language is decidable if its complement is decidable.
- A language is not Turing recognizable.
Notes on EQM:

- **EQM** is Turing-recognizable (idea: we build an enumerator for \( \Sigma^* \) and we give as input the words it builds up to \( M \); we stop when a word is finally accepted).

**EQM** and **EQM** are not Turing-recognizable.

To prove that \( B \) is not Turing-recognizable we may show that \( A_{TM} \subseteq B \).

**Chap 6: Time Complexity**

**MEASURING COMPLEXITY**

- Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory. In this final part of the book, we introduce computational complexity theory — an investigation of the time, memory, or other resources required for solving computational problems.

- For simplicity we compute the running time of an algorithm purely as a function of the length of the string representing the input and don't consider any other parameters. In worst case analysis, the form we consider here, we consider the longest running time of all inputs of a particular length.

**Definition**

Let \( M \) be a deterministic Turing machine that halts on all inputs. The running time or time complexity of \( M \) is the function \( f(n) \), where \( f(n) \) is the maximum number of steps that \( M \) uses on any input of length \( n \). If \( f(n) \) is the running time of \( M \) we say that \( M \) runs in time \( f(n) \) and that \( M \) is an \( f(n) \) time Turing machine.

- Because the exact running time of an algorithm often is a complex expression, we usually estimate it, i.e. in a convenient form of estimation, called asymptotic analysis, we seek to understand the running time of the algorithm when it is run on large inputs.

**Definition** (Big-\( O \) notation) 

- if \( f(n) \) is a function, where \( f(n) \) is an integer for \( n \), then \( f(n) \) is the maximum number of steps that \( M \) uses on any input of length \( n \) and if \( f(n) \) is the running time of \( M \) we say that \( M \) runs in time \( f(n) \) if \( f(n) \) is an \( f(n) \) time Turing machine.

- Because the exact running time of an algorithm often is a complex expression, we usually estimate it, i.e. in a convenient form of estimation, called asymptotic analysis, we seek to understand the running time of the algorithm when it is run on large inputs.

**Definition** (Smaller notation) 

If \( f(n) \) and \( g(n) \) be two functions \( f(n) < g(n) \) if \( f(n) \) is the maximum number of steps that \( M \) uses on any input of length \( n \) and if \( f(n) \) is the running time of \( M \) we say that \( M \) runs in time \( f(n) \) if \( f(n) < g(n) \) and that \( M \) is an \( f(n) \) time Turing machine.

**Definition** (Time complexity class) 

- If \( f(n) \) is a function, where \( f(n) \) is an integer for \( n \), then \( f(n) \) is the maximum number of steps that \( M \) uses on any branch \( \ell \) its computation on any input of length \( n \).

**THE CLASS \( P \)**

- **Exponential time algorithms typically arise when we solve problems by searching through a space of solutions (brute force).**

- **The class \( P \) plays a central role in our theory and is important because:**
  - \( P \) is important for all models of computation that are polynomial equivalent to the deterministic single-tape Turing machine.
  - \( P \) roughly corresponds to the class of problems that are realistically solvable on a
P versus NP

Notes on NP:

• P = {L | L is decided by a deterministic TM in polynomial time.}
• NP = {L | L is decided by a non-deterministic TM in polynomial time.}
• P ⊆ NP
• EXP = {L | L is decided by a deterministic TM in exponential time}
• NP ⊆ EXP

The Class NP

Examples of problems in P:
1. Every context-free language is a member of P.
2. P is closed under union, intersection, and complement.

Example of problems in NP:
1. PATH = {<G,w> | G is a directed graph with a directed path from s to t} is in P.
2. A polynomial time algorithm for PATH is given by TM M with input <G,w> behaves as follows:
   1. Place a mark on node s.
   2. Repeat the stage 3 until no additional nodes are marked:
      3. Scan all edges of G. If an edge (u,v) is found going from a marked node u to an unmarked node v, mark v.
   4. If it is marked, accept. Otherwise, reject.

3. NP means Non-Deterministic Polynomial.
4. Hamilton - P = HAMPATH = {<G,w> | G is a directed graph with a Hamiltonian path from s to t}.
5. The HAMPATH problem does have a feature called polynomial verifiability.
6. Some problems may not be polynomial verifiable. For example, take HAMPATH, the complement of the HAMPATH problem. Even if we could determine (somewhat) that a graph did not have a Hamiltonian path, we don’t know of a way for someone else to verify its non-existence without using the same exponential time algorithm for making the determination in the first place.
7. A verifier for a language A is an algorithm V, where A = {w | V accepts <w,c> for some string c}.
8. We measure the time of a verifier only in terms of the length of w so a polynomial time verifier runs in polynomial time in the length of w.

NP is the class of languages that have polynomial time verifiers.

Notes on NP:

1. P versus NP:
   - P = class of languages for which membership can be verified quickly.
   - NP = class of languages for which membership can be verified quickly over a non-deterministic tape.

2. Every context-free language is a member of P.
3. P is closed under union, intersection, and complement.

3. SAT = {<B> | B is a satisfiable Boolean formula} is in NP.
4. Cook - Levin theorem: SAT ∈ P NP.

5. We have already defined the concept of reducing one problem to another. When problem B reduces to problem A, a solution to B can be used to solve A. Now we define a version of reducibility that takes the efficiency of the algorithm into account. When problem B is efficiently reducible to problem A, we say that B is polynomial time reducible to A.

6. Definition: A polynomial time computable function f: Σ* → Σ* is a polynomial time computable function if for every w in Σ*, the function f(w) can be computed in polynomial time.

7. Definition: A language A is polynomial time mapping reducible, or simply polynomial time reducible, to language B, written A ≤P B, if A is a polynomial time computable function f: Σ* → Σ* exists, where for every w in Σ*, f(w) is in B, and f can be computed in polynomial time.

8. SAT is polynomial time reducible to CLIQUE. This means, if CLIQUE is polynomial time solvable, so is SAT.

9. Definition: A language B is in NP-complete if it is polynomial time reducible to all other languages in NP.

10. HAMPATH is NP-hard. HAMPATH ≤P HAMCIRCUIT (new vertex added to the graph).
11. SAT is NP-complete (Cook-Levin theorem).
12. SAT is NP-complete (Cook-Levin theorem).

RELPRIME = {<x,y> | x and y are relatively prime} is in P.
CONNECTED = {<G> | G is a connected undirected graph} is in P.

Proof idea: Same demonstration as PATH but with an additional step to check for connectedness.

TRIANGLE, 3-CLIQUE, 4-CLIQUE, and 5-CLIQUE are in P. But we caution: is 1-CLIQUE (or 3) is in NP?! Proof idea for triangle: Select all subsets of 3 vertices and check if they are connected. Graphs of size n => O(n!/(n-k)! * k!) = O(n^2).

A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of A. This information is called a certificate, or proof, of membership in A. Observe that, for polynomial verifiers, the certificate has polynomial length. (In the length of w) because that is how the verifier can access its time bound.

A language A is in NP if it is decided by some nondeterministic polynomial time Turing machine.

NTIME(t(n)) = {L | L is decided by a O(t(n)) time nondeterministic Turing machine}

NP = ∪t NTIME(t(n))

PSPACE = {L | L is decided by a polynomial space non-deterministic Turing machine.

The best known method for solving problems in NP deterministically uses exponential time algorithm: NP ⊆ EXP = {L | L is decided by a deterministic TM in exponential time}.

SUBSET-SUM = {<S,t> | S = {x_k,...,x_1} and for some (y_1,...y_k) ∈ {x_k,...,x_1}, we have Σy_1=1}

NP and coNP:
1. The class coNP contains the languages for which the complement is in NP.
2. HAMPATH, CLIQUE, and SUBSET-SUM are not obviously also in NP. Verifying that something is not present seems to be more difficult than verifying that it is present.
3. We believe that NP ≠ coNP, but clearly P ≠ coNP.

V. CLIQUE is NP-complete. Proof idea for the reduction of 3SAT to CLIQUE.

The nones of G are organized in k groups of three nodes each. Each group contains one of the clauses in φ, and each node in a triple corresponds to a literal in the clause. Label each node of G with its corresponding literal in φ. The edges of G connect all but 2 types of pairs of nodes: No edge is present between nodes in the same triple and no edge is present between a nodes in the same triple and no edge is present between two nodes with contradictory labels, i.e., in X2. The clause is satisfiable if the graph has a clique.

6. If G is an undirected graph, a vertex cover of G is a subset of the nodes where every edge of G touches at least one of those nodes. The vertex cover problem asks for the size of the smallest vertex cover.

7. VERTEX-COVER = {<G> | G is an undirected graph that has a k - node vertex cover}.

8. VERTEX-COVER is NP-complete. Proof idea: Reduction of 3SAT to VERTEX-COVER.

9. Each variable in φ produces two variables labeled x and y, and each clause produces three variables labeled x, y, and z, which are all connected together.

10. Each node v produces in L is connected to all nodes in G. Each node produced in L is connected to all nodes in G. There are 2m+3n clauses in G (with m variables and n clauses) that we look for a k = m + 2l vertex cover.

11. HAMPATH is NP-complete. HAMPATH ≤P HAMCIRCUIT (new vertex added to the graph).

12. SUBSET-SUM is NP-complete. HAMCIRCUIT ≤P TSP (HAMCIRCUIT is a specific case of TSP).