At the end of the course, students should be able to

(1) devise new algorithms using
   a. the divide and conquer pattern
   b. dynamic programming
   c. the greedy approach
(2) prove the correctness of algorithms using loop invariants
(3) analyze the efficiency of algorithms using
   a. worse-case and expected-case analysis
   b. recurrences and the substitution method
   c. the master method
   d. amortized analysis
(4) **articulate known algorithmic results**
(5) **prove the undecidability or intractability of a problem**

Under *Computational Complexity*: We study the theory of computation with the goal that students learn the terminology and concepts of the field, the important results, and how to do reductions as used in proofs of undecidability and intractability.

A. Languages and automata (LP ch 2-4)

   1. Finite automata and regular expressions (LP ch 2)
   2. Context-free languages (LP ch 3)
   3. Turing machines (LP 4.(1-4))
   4. Non-deterministic Turing machines (LP 4.5)

B. Undecidablility (LP ch 5)

   1. Definition of undecidablility (LP 5.(1-3))
   2. Undecidability proofs (LP 5.(4-7))

C. $\mathcal{NP}$-completeness (LP ch 6 & 7 and CLRS ch 34)

   1. The classes $\mathcal{P}$ and $\mathcal{NP}$ (LP ch 6)
   2. $\mathcal{NP}$-completeness proofs (LP ch 7)
   3. $\mathcal{NP}$-complete problems (CLRS ch 34)
   4. Perspectives on $\mathcal{NP}$-completeness

To say that a problem "can be solved in polynomial time" means

> *For this problem there exists and algorithm such that for any instance X of that problem, the algorithm solves that instance of the problem in worst case $O(f)$ where f is a function that is polynomial in $|X|$.*

Let $\mathcal{P}$ (different from $\mathscr{P}$) denote the set of such *problems*.

(Be sure you can keep straight the difference among *problems*, *instances of problems*, and *algorithms*.)

There exist some problems such that, if we are given *an instance of that problem* and *a candidate result/output for that instance*, we can verify that candidate in polynomial time.

That is, there exists and algorithm that takes a problem instance and a (possible) solution and determines whether that the given possible solution meets the requirements for being a solution to the problem instance.

Let $\mathcal{NP}$ denote the set of such problems.

There exist some $\mathcal{NP}$ problems such that we can convert any $\mathcal{NP}$ problems to them. Call these problems $\mathcal{NP}$-complete.

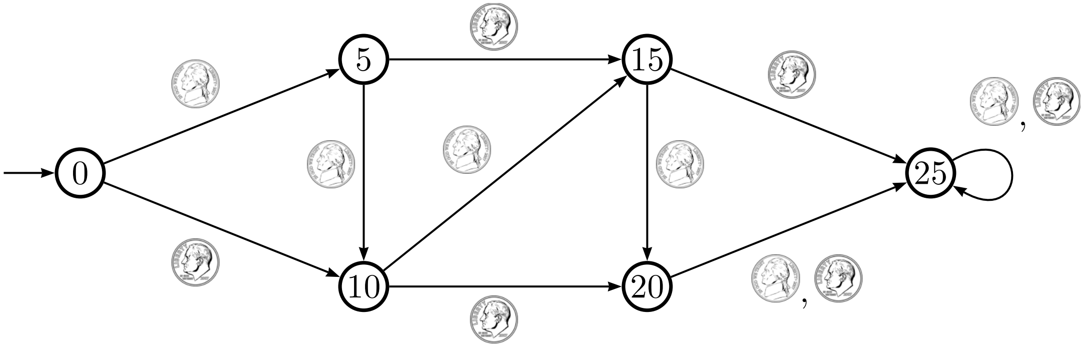
That is,

> *Given a problem P_1 known to be in the set $\mathcal{NP}$ and a problem P_2 known to be $\mathcal{NP}$-complete, there exists an algorithm to convert any instance of P_1 to an instance P_2, and a corresponding algorithm to convert the result of solving the instance of P_2 back to a result for the instance of P_1.*

If *any* $\mathcal{NP}$-complete problem is solvable in polynomial time, then *all* $\mathcal{NP}$ problems are solvable in polynomial time, and $\mathcal{P} = \mathcal{NP}$.

Contrapositively, if $\mathcal{P} \neq \mathcal{NP}$, then *no* $\mathcal{NP}$-complete problem is solvable in polynomial time.

$$\begin{tikzcd}
{} \arrow[r] & O \arrow[r, "pull"] & S \arrow[r, "pull"] & M \arrow[r, "pull"] & F \arrow[lll, bend left, "pull"]
\end{tikzcd}$$



Diagrams generated by VAUCANSON-G

Definition 2.1.1: A **deterministic finite automaton (DFA)** is a quintuple $M = (K, \Sigma, \delta, s, F)$ where

- K is a finite set of **states**
- Σ is an alphabet
- $s \in K$ is the **initial state**
- $F \subseteq K$ is the set of **final states**
- δ, the **transition function** is a function from $K \times \Sigma$ to K.

LP pg 57

A **configuration** is the “state” of the machine: the (formal) state plus the remaining string: $K \times \Sigma*$

$\vdash_M$ is a relation on $K \times \Sigma*$, that is, a relation from configurations to configurations.

$(q_1, \mathtt{a}w) \vdash_M (q_2, w)$ if $\delta(q_1, a) = q_2$.

The language accepted by DFA M is

$$L(M) = \{\ w \in \Sigma* \mid (s, w) \vdash_M *(f, \varepsilon) \text{ where } f \in F\ \}$$

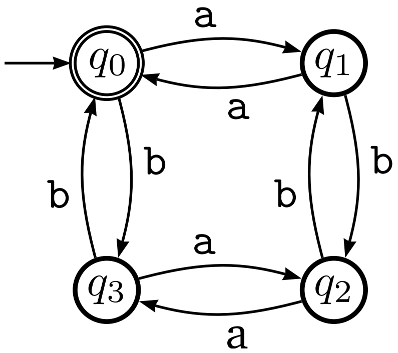


Diagram generated by VAUCANSON-G

Ex 2.1.5.b. Formally define a deterministic 2-tape finite automaton, the notion of a configuration, etc.

$$M = (K_1, K_2, \Sigma, \Delta, s, F)$$

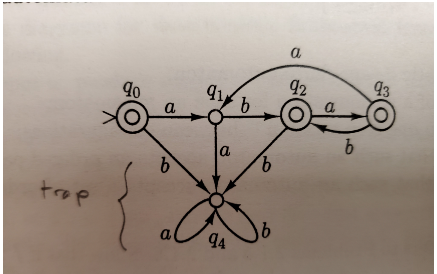


Figure 2-4

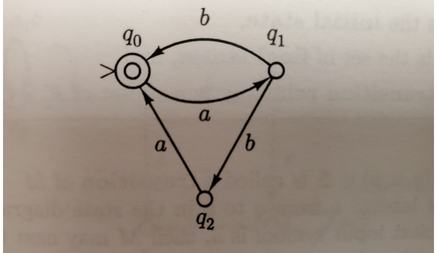


Figure 2-5

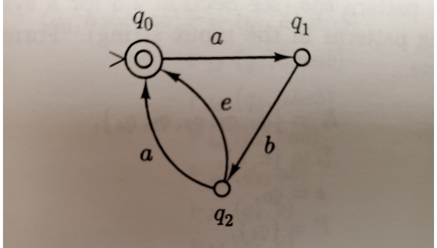


Figure 2-6

Definition 2.1.1: A **nondeterministic finite automaton (NFA)** is a quintuple $M = (K, \Sigma, \Delta, s, F)$ where

- K is a finite set of **states**
- Σ is an alphabet
- $s \in K$ is the **initial state**
- $F \subseteq K$ is the set of **final states**
- Δ, the **transition relation** is a relation from $K \times (\Sigma \cup \{\varepsilon\})$ to K.

LP pg 65–66

Three ways to imagine (“intuit”) non-determinism

- The machine oracularly always guesses the right step.
- The machine tries something, and if that doesn’t yield acceptance, backs up and tries again, until finding a walk to an accept state or exhausts all possibilities.
- The machine forks a thread, and each thread tries a different path.

Class (of machine) equivalence:

> *It is therefore evident that the class of languages accepted by deterministic automata is a subset of the class of languages accepted by nondeterministic automata.* (LP 68)

Machine equivalence:

> *Formally, we say that two finite automata M_1 and M_2 (deterministic or nondeterministic) are* **equivalent** *if and only if (sic) $L(M_1) = L(M_2)$. Thus two automata are considered to be equivalent if they accept the same language, even though they may "use different methods" to do so.* (LP 69)

Consider the difference between

- The set of strings (*language*) accepted by a machine M:

$$L(M) = \{\ w \in \Sigma* \mid (s, w) \vdash_M *(f, \varepsilon) \text{ where } f \in F\ \}$$

- The set of languages accepted by DFAs as a class of machines:

$$\{L \mid \exists\ M \in DFA \text{ such that } L = L(M)\}$$

“Obviously” the set of languages accepted by DFAs is a subset of the set of languages accepted by NFAs. . .

Theorem 2.2.1: For each nondeterministic finite automaton, there is an equivalent deterministic automaton.

$$\forall\ M \in NFA,\ \exists\ M' \in DFA \text{ such that } L(M) = L(M')$$

Outline of the *constructive proof*:

```
Suppose $M \in NFA$
    Construct $M' \in DFA$
    Prove $L(M) = L(M')$, that is,
        Suppose $w \in \Sigma*$
        Prove either
            Both $M$ and $M'$ take $w$ into an accept state, or
            neither $M$ nor $M'$ take $w$ into an accept state
```

Theorem 2.2.1: For each nondeterministic finite automaton, there is an equivalent deterministic automaton.

$$\forall\ M \in NFA,\ \exists\ M' \in DFA \text{ such that } L(M) = L(M')$$

Outline of the *constructive proof*:

```
Suppose $M \in NFA$
    Construct $M' \in DFA$
    Show that $M'$ is deterministic
    Prove $L(M) = L(M')$, that is,
        Suppose $w \in \Sigma*$
        Prove $(q, w) \vdash_M *(p, \varepsilon)$ iff $(E(q), w) \vdash_{M'} *(P, \varepsilon)$
            for some set $P$ such that $p \in P$.
        (Proof by induction on the length of $w$)
```