Ch1.1 (Week 5)

Introduction

- The rules of logic are used to distinuish between valid and invalid mathematical arguments.

Propositions

- A **proposition** is a sentence that declares a fact, that is either true or false, but not both.

1 Let p be a proposition. The *negation* of p, denoted by $\neg p$ (also denoted by $\overline{p}$), is the statement "It is not the case that p."
The proposition $\neg p$ is read "not p." The truth value of the negation of p, $\neg p$, is the opposite of the truth value p.

TABLE 1 The Truth Table for the Negation of a Proposition.

p	$\neg p$
T	F
F	T

2 Let p and q be propositions. The *conjunction* of p and q, denoted by $p \wedge q$, is the proposition "p and q." The conjunction $p \wedge q$ is true when both p and q are true and is false otherwise.

TABLE 2 The Truth Table for the Conjunction of Two Propositions.

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

- In logic, the word "but" sometimes is used instead of "and" in a conjunction.

3 Let p and q be propositions. The *disjunction* of p and q, denoted by $p \vee q$, is the proposition "p or q." The conjunction $p \vee q$ is true when both p and q are false and is true otherwise.

TABLE 3 The Truth Table for the Disjunction of Two Propositions.

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

4 Let p and q be propositions. The *exclusive or* of p and q, denoted by $p \oplus q$, is the proposition that is true when exactly one of p and q is true and is false otherwise.

TABLE 4 The Truth Table for the Exclusive Or of Two Propositions.

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

Conditional Statements (Implication)

5 Let p and q be propositions. The *conditional statement* $p \rightarrow q$ is the proposition "if p , then q." The conditional statement $p \rightarrow q$ is false when p is true and q is false, and true otherwise.
In the conditional statement $p \rightarrow q$, p is called the *hypothesis* (or *antecedent* or *premise*) and q is called the *conclusion* (or *consequence*).

TABLE 5 The Truth Table for the Conditional Statement $p \rightarrow q$.

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

- Common ways to express conditional statement:
  - "if p, then q." "if p, q." "q if p" "q when p" "p implies q" "p only if q"

Converse, Contrapositive, and **Inverse**

$$
\begin{aligned}
p &\to q \\
q &\to p && converse \\
\neg q &\to \neg p && contrapositive \\
\neg p &\to \neg q && inverse
\end{aligned}
$$

- *Equivalent* (truth value)
  Origin == Contrapositive
  Converse == Inverse

6 Let p and q be propositions. The *biconditional statement* $p \leftrightarrow q$ is the proposition "if p and only if q." The biconditional statement $p \leftrightarrow q$ is true when p and q have the same truth values, and is false otherwise.
Biconditional statements are also called *bi-implications*.

TABLE 6 The Truth Table for the Biconditional $p \leftrightarrow q$.

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

- Common ways to express biconditional statement:
  - "p *iff* q."
- Same truth value as $(p \to q) \wedge (q \to p)$

Truth Tables and Compound Propositions

TABLE 7 The Truth Table of $(p \vee \neg q) \to (p \wedge q)$.

p	q	$\neg q$	$p \vee \neg q$	$p \wedge q$	$(p \vee \neg q) \to (p \wedge q)$
T	T	F	T	T	T
T	F	T	T	F	F
F	T	F	F	F	T
F	F	T	T	F	F

Precedence of Logical Operators

TABLE 8 Precedence of Logical Operators.

Operator	*Precedence*
$\neg$	1
$\wedge$	2
$\vee$	3
$\to$	4
$\leftrightarrow$	5

Logic and Bit Operations

Truth Value	*Bit*
T	1
F	0

TABLE 9 Table for the Bit Operators *OR*, *AND*, and *XOR*.

x	y	$x \vee y$	$x \wedge y$	$x \oplus y$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

7 A *bit strings* is a sequence of zero or more bits. The *length* of this string is the number of bits in the string.

- bitwise notation:
  - OR: $\vee$, AND: $\wedge$, XOR: $\oplus$

Ch1.4 (Week 7)

Predicate

Propositional function P

- A statement of the form $P(x_1, x_2, ..., x_n)$, also called n-**place predicate** or a n-**ary predicate**.

Preconditions & Postconditions

- Valid input & satisfied output

▼ EX.7

`temp := x` `x := y` `y := temp` Find the predicate that can use as the precondition and postcondition.

Precondition → $P(x, y)$ is "$x = a$ and $y = b$"
Postcondition → $Q(x, y)$ is "$x = b$ and $y = a$"

To Verify:
Assume the precondition holds, go through the first step `temp := x`, then $x = a$, $temp = a$, and $y = b$. After the second step `x := y`, then $x = b$, $temp = a$, and $y = b$. Finally, after the third step `y := temp`, then $x = b$, $temp = a$, and $y = a$. After the program is run, the postcondition $Q(x, y)$ holds.

Quantifiers

Predicate Calculus

- The area of logic that deals with predicates and quantifiers

Domain | | Domain of Discourse

- a property is true for all values of a variable in a particular domain

1 The *universal quantification* of $P(x)$ is the statement
"$P(x)$ for all values of x in the domain."
The notation $\forall x P(x)$ denotes the universal quantification of $P(x)$. Here $\forall$ is called the universal quantifier. We read $\forall x P(x)$ as "for all $xP(x)$" or "for every $xP(x)$." An element for which $P(x)$ is false is called a counterexample of $\forall x P(x)$.

- "all of", "for each", "given any", "for arbitrary", and "for any"

TABLE 1 Quantifiers.

Statement	*When True?*	*When False?*
$\forall x\, P(x)$	$P(x)$ is true for every x.	There is an x for which $P(x)$ is false.
$\exists x\, P(x)$	There is an x for which $P(x)$ is true.	$P(x)$ is false for every x.

2 The *existential quantification* of $P(x)$ is the proposition
"There exists an element x in the domain such that $P(x)$."
We use the notation $\exists x P(x)$ for the existential quantification of $P(x)$. Here $\exists$ is called the existential quantifier.

- "for some", "for at least one", or "there is"
- Without specifying the domain, the statement $\exists x P(x)$ has no meaning

The Uniqueness Quantifier

- denoted by $\exists!$ or $\exists_1$

  "There exist a unique x such that $P(x)$ is true.", "there is exactly one" and "there is one and only one"

Quantifiers and Restricted Domains

- $\forall x < 0\ (x^2 > 0)$ is the same as $\forall x(x < 0 \rightarrow x^2 > 0)$ [Conditional Statement]
- $\exists z > 0\ (x^2 = 2)$ is the same as $\exists z\ (x > 0 \wedge z^2 = 2)$ [Conjunction]

Precedence of Quantifiers

- $\forall$ and $\exists$ have higher precedence than all logical operators

  $\forall x P(x) \vee Q(x)$ means $(\forall x\ P(x)) \vee Q(x)$ rather than $\forall x(P(x) \vee Q(x))$

Binding Variables

Scope

- the part of a logical expression that a quantifier is applied
- free variable: outside the scope

▼ EX.18

In the statement $\exists x\ (x + y = 1)$, the variable x is bound by the existential quantification $\exists x$, but the variable y is free because it is not bound by a quantifier and no value is assigned.

⇒ x is bound, but y is free.

Logical Equivalences Involving Quantifiers

3 Statements involving predicates and quantifiers are *logically equivalent* if and only if they have the same truth value no matter which predicates are substituted into these statements and which domain of discourse is used for the variables in these propositional functions. We use the notation $S \equiv T$ to indicate that two statements S and T involving predicates and quantifiers are logically equivalent.

Negating Quantified Expressions

▼ example

$P(x)$ is "x has taken a course in calculus", domain → students in your class.
Negation → "It is not the case that every student in your class has taken a course in calculus."
Equivalent to → "There is a student in your class who has not taken a course in calculus."

- $\neg\forall x\ P(x) \equiv \exists x\ \neg P(x)$
- $\neg\exists x\ Q(x) \equiv\ \forall x\ \neg Q(x)$

TABLE 2 De Morgan's Laws for Quantifiers.

Negation	*Equivalent Statement*	*When Is Negation True?*	*When False?*
$\neg\exists x\, P(x)$	$\forall x \neg P(x)$	For every x, $P(x)$ is false.	There is an x for which $P(x)$ is true.
$\neg\forall x\, P(x)$	$\exists x \neg P(x)$	There is an x for which $P(x)$ is false.	$P(x)$ is true for every x.

Ch 1.3 & 1.6 (Week 8)

Propositional Equivalences

Introduction

Compound proposition

- An expression formed from propositional variables using logical operators

1. A compound proposition that is always true, no matter what the truth values of the propositional variables that occur in it, is called a *tautology*. A compound proposition that is always false is called a *contradiction*. A compound proposition that is neither a tautology nor a contradiction is called a *contingency*.

Logical Equivalences

- Have the same truth values in all possible cases

2. The compound propositions p and q are called *logically equivalent* if $p \leftrightarrow q$ is a tautology. The notation $p \equiv q$ denotes that p and q are logically equivalent.

- $\Leftrightarrow$ is sometimes used instead of $\equiv$

Rules of Inference

Introduction

Argument : A sequence of statements that end with a conclusion.

Valid : The conclusion, or final statement of the argument, must follow from the truth of the preceding statements, or premises, of the argument.

Valid Arguments in Propositional Logic

1. An *argument* in propositional logic is a sequence of propositions. All but the final proposition in the argument are called *premises* and the final proposition is called the *conclusion*. An argument is *valid* if the truth of all its premises implies that the conclusion is true.

   An *argument form* in propositional logic is a sequence of compound propositions involving propositional variables. An argument form is *valid* no matter which particular propositions are substituted for the propositional variables in its premises, the conclusion is true if the premises are all true.

Rules of Inference for Propositional Logic

TABLE 1 Rules of Inference.

Rule of Inference	*Tautology*	*Name*
p $p \to q$ $\therefore q$	$(p \wedge (p \to q)) \to q$	Modus ponens
$\neg q$ $p \to q$ $\therefore \neg p$	$(\neg q \wedge (p \to q)) \to \neg p$	Modus tollens
$p \to q$ $q \to r$ $\therefore p \to r$	$((p \to q) \wedge (q \to r)) \to (p \to r)$	Hypothetical syllogism
$p \vee q$ $\neg p$ $\therefore q$	$((p \vee q) \wedge \neg p) \to q$	Disjunctive syllogism
p $\therefore p \vee q$	$p \to (p \vee q)$	Addition
$p \wedge q$ $\therefore p$	$(p \wedge q) \to p$	Simplification
p q $\therefore p \wedge q$	$((p) \wedge (q)) \to (p \wedge q)$	Conjunction
$p \vee q$ $\neg p \vee r$ $\therefore q \vee r$	$((p \vee q) \wedge (\neg p \vee r)) \to (q \vee r)$	Resolution

▼ Example 3

"It is below freezing now. Therefore, it is either below freezing or raining now."

Solution:
p is "It is below freezing now", and q is "It is raining now."

$$\frac{p}{\therefore\ p \vee q}$$

This is an argument that uses the *addition rule*.

▼ Example 4

"It is below freezing and raining now. Therefore, it is below freezing now."

Solution:
p is "It is below freezing now", and q is "It is raining now."

$$\frac{p \wedge q}{\therefore\ p}$$

This argument uses the *simplification rule*.

▼ Example 5

"If is rains today, then we will not have a barbecue today. If we do not have a barbecue today, then we will have a barbecue tomorrow. Therefore, if it rains today, we will have a barbecue tomorrow."

Solution:
p is "It is raining today," q is "We will not have a barbecue today," and r is "We will have a barbecue tomorrow."

$$\frac{\begin{array}{l} p \to q \\ q \to r \end{array}}{\therefore\ p \to r}$$

Hence, this argument is a *hypothetical syllogism*.

Using Rules of Inference to Build Arguments

▼ Example 7

"If you send me an email message, then I will finish writing the program," "If you do not send me an email message, then I will go to sleep early," and "If I go to sleep early, then I will wake up feeling refreshed" lead to the conclusion "If I do not finish writing the program, then I will wake up feeling refreshed."

Solution:
p is "You sent me an email message", q is "I will finish writing the program", r is "I will go to sleep early", and s is "I will wake up feeling refreshed".
The premises are $p \to q$, $\neg p \to r$, and $r \to s$. The desired conclusion is $\neg q \to s$.

Step	**Reason**
1. $p \to q$	Premise
2. $\neg q \to \neg p$	Contrapositive of (1)
3. $\neg p \to r$	Premise
4. $\neg q \to r$	Hypothetical syllogism using (2) and (3)
5. $r \to s$	Premise
6. $\neg q \to s$	Hypothetical syllogism using (4) and (5)

Resolution

Resolution → $((p \vee q) \wedge (\neg p \vee r)) \to (q \vee r)$

Resolvent → $(q \vee r)$

Clauses → A disjunction of variables or negations of these variables.

Fallacies

$((p \to q) \wedge q) \to p$ is not a tautology ⇒ fallacy of affirming the conclusion

$((p \to q) \wedge \neg p) \to \neg q$ is not a tautology ⇒ fallacy of denying the hypothesis

Rules of Inference for Quantified Statements

TABLE 2 Rules of Inference for Quantified Statements.

Rule of Inference	*Name*
$\dfrac{\forall x\, P(x)}{\therefore\ P(c)}$	Universal instantiation
$\dfrac{P(c) \text{ for an arbitrary } c}{\therefore\ \forall x\, P(x)}$	Universal generalization
$\dfrac{\exists x\, P(x)}{\therefore\ P(c) \text{ for some element } c}$	Existential instantiation
$\dfrac{P(c) \text{ for some element } c}{\therefore\ \exists x\, P(x)}$	Existential generalization

Universal instantiation

$P(c)$ is true, where c is a particular member of the domain, given the premise $\forall x P(x)$.

Universal generalization

$\forall x P(x)$ is true, given the premise that $P(c)$ is true for all elements c in the domain.

Existential instantiation

There is an element c in the domain for which $P(c)$ is true if we know that $\exists x P(x)$ is true.

Existential generalization

$\exists x P(x)$ is true when a particular element c with $P(c)$ true is known.

Combining Rules of Inference for Propositions and Quantified Statements

Universal modus ponens ⇒ universal instantiation + modus penens

if $\forall x(P(x) \to Q(x))$ is true, and if $P(a)$ is true for a particular element a in the domain of the universal quantifier, then $Q(a)$ must also be true.

Universal modus tollens ⇒ univesal instantiation + modus tollens

if $\forall x(P(x) \to Q(x))$ is true, and $\neg Q(a)$ is true where a is a particular element in the domain, then $\neg P(a)$ must be true.

Ch6.1 The Basics of Counting

Basic Counting Principles

THE PRODUCT RULE Suppose that a procedure can be broken down into a sequence of two tasks. If there are n_1 ways to do the first task and for each of these ways of doing the first task, there are n_2 ways to do the second task, then there are $n_1 n_2$ ways to do the procedure.

$|A_1 \times A_2 \times \cdots \times A_m| = |A_1| \cdot |A_2| \cdot \cdots \cdot |A_m|$.

▼ Example 5

How many different license plates can be made if each plate contains a sequence of three uppercase English letters followed by three digits (and no sequences of letters are prohibited)?

Solution: There are 26 choices for each of the three uppercase English letters and ten choices for each of the three digits. Hence, by the product rule there are a total of $26 \cdot 26 \cdot 26 \cdot 10 \cdot 10 \cdot 10 = 17,576,000$ possible license plates.

THE SUM RULE If a task can be done either in one of n_1 ways or in one of n_2 ways, where none of the set of n_1 ways is the same as any of the set of n_2 ways, then there are $n_1 + n_2$ ways to do the task.

$|A_1 \cup A_2 \cup \cdots \cup A_m| = |A_1| + |A_2| + \cdots + |A_m|$ when $A_i \cap A_j = \forall i, j$.

▼ Example 13

A student can choose a computer project from one of three lists. The three lists contain 23, 15, and 19 possible projects, respectively. No project is one more then one list. How many possible projects are there to choose from?

Solution: The student can choose a project by selecting a project from the first list, the second list, or the third list. Because no project is on more than one list, by the sum rule there are 23+15+19=57 ways to choose a project.

More Complex Counting Problems

▼ Example 15

In a version of the computer language BASIC, the name of a variable is a string of one or two alphanumeric characters, where uppercase and lowercase letters are not distinguished. (An *alphanumeric* character is either one of the 26 English letters or one of the 10 digits.) Moreover, a variable name must begin with a letter and must be different from the five strings or two characters that are reserved for programming use. How many different variable names are there in this version of BASIC?

Solution: Let V equal the number of different variable names in this version of BASIC. Let V_1 be the number of these that are one character long and V_2 be the number of these that are two characters long. Then by the sum rule, $V = V_1 + V_2$. Note that $V_1 = 26$, because a one-character variable name must be a letter. Furthermore, by the product rule there are $26 \cdot 36$ strings of length two that begin with a letter and end with an alphanumeric character. However, five of these are excluded, so $V_2 = 26 \cdot 36 - 5 = 931$. Hence, there are $V = V_1 + V_2 = 26 + 931 = 957$ different names for variables in this version of BASIC.

The Subtraction Rule (Inclusion-Exclusion for Two Sets)

THE SUBTRACTION RULE If a task can be done in either n_1 ways or n_2 ways, then the number of ways to do the task is $n_1 + n_2$ minus the number of ways to do the task that are common to the two different ways.

$|A_1 \cup A_2| = |A_1| + |A_2| - |A_1 \cap A_2|$.

▼ Example 18

How many bit strings of length eight either start with a 1 bit or end with the two bits 00?

Solution: We can construct a bit string of length either that begins with a 1 in $2^7 = 128$ ways. Similarly, we can construct a bit string of length eight ending with the two bits 00, in $2^6 = 64$ ways. Some of the ways to construct a bit string of length eight starting with a 1 are the same as the ways to construct a bit string of length eight that ends with the two bits 00. There are $2^5 = 32$ ways to construct such a string. Consequently, the number of bit strings of length eight that begin with a 1 or end with a 00, which equals the number of ways to construct a bit string of length eight that begins with a 1 or that ends with 00, equals 128+64-32=160.

The Division Rule

THE DIVISION RULE There are n/d ways to do a task if it can be done using a procedure that can be carried out in n ways, and for every way w, exactly d of the n ways correspond to way w.

$\rightarrow$ "If the finite set A is the union of n pairwise disjoint subsets each with d elements, then $n = |A|/d$."

▼ Example 20

How many different ways are there to seat four people around a circular table, where two seatings are considered the same when each person has the same left neighbor and the same right neightbor?

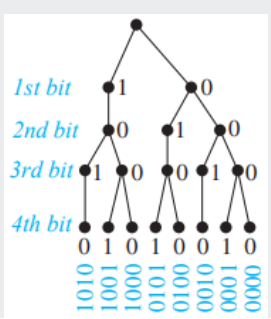
Solution: We arbitrarily select a seat at the table and label it seat 1. We number the rest of the seats in numerical order, proceeding clockwise around the table. Note there are four ways to select the person for seat 1, three for seat 2, two for seat 3, and one for seat 4. Thus, there are $4! = 24$ ways to order the given four people for these seats. However, each of the four choices for seat 1 leads to the same arrangement, as we distinguish two arrangements only when one of the people has a different immediate left or immediate right neighbour. Because there are four ways to choose the person for seat 1, by the division rule there are $24/4 = 6$ different seating arrangements of four people around the circular table.

Tree Diagrams

▼ Example 21

How many bit strings of length four do not have two consecutive 1s?

Solution: The tree diagram displays all bit strings of length four without two consecutive 1s. We see that there are eight bit strings of length four without two consecutive 1s.



Ch6.3 Permutations and Combinations

Permutations

1 **THEOREM 1** If n is a positive integer and r is an integer with $1 \leq r \leq n$, then there are $P(n, r) = n(n-1)(n-2)\cdots(n-r+1)$ r-permutations of a set with n distinct elements.

1 **COROLLARY 1** If n and r are integers with $0 \leq r \leq n$, then $P(n, r) = \dfrac{n!}{(n-r)!}$.

▼ Example 5

Suppose there are 8 runners in a race. The winner receives a gold medal, the second-place finisher receives a silver medal, and the third-place finisher receives a bronze medal. How many different ways are there to award these medals, if all possible outcomes of the race can occur and there ar

Combinations

2 **THEOREM 2** The number of r-combinations of a set with n elements, where n is a nonnegative integer and r is an integer with is an integer with $0 \leq r \leq n$, equals $C(n, r) = \dfrac{n!}{r!\,(n-r)!}$.

2 **COROLLARY 2** Let n and r be nonnegative integers with $r \leq n$. Then $C(n, r) = C(n, n-r)$.

▼ Example 11

How many poker hands of give cards can be dealt from a standard deck of 52 cards? Also, how many ways are there to select 47 cards from a standard deck of 52 cards?

Solution: Because the order in which the five cards are dealt from a deck of 52 cards does not matter, there are $C(52, 5) = \dfrac{52!}{5!47!} = \dfrac{52 \cdot 51 \cdot 50 \cdot 49 \cdot 48}{5 \cdot 4 \cdot 3 \cdot 2 \cdot 1} = 26 \cdot 17 \cdot 10 \cdot 49 \cdot 12 = 2,598,960$ different hands of five cards that can be dealt. Consequently, there are $C(52, 47) = \dfrac{52!}{47!5!} = 2,598,960$ different poker hands of give cards that can be dealt from a standard deck of 52 cards.

1 A *combinatorial proof* of an identity is a proof that uses counting arguments to prove that both sides of the identity count the same objects but in different ways or a proof that is based on showing that there is a bijection between the sets of objects counted by the two sides of the identity. These two types of proofs are called *double counting proofs* and *bijective proofs*, respectively.

Ch6.5 Generlized Permutations and Combinations

Permutations with Repetition

1 THEOREM 1 The number of r-permutations of a set of n objects with repetition allowed is n^r.

Combinations with Repetition

▼ Example 3

How many ways are there to select 5 bills from a cash box containing \$1, \$2, \$5, \$10, \$20, \$50, and \$100 bills? Assume that the order in which the bills are chosen does not matter, that the bills of each denomination are indistinguishable, and that there are at least 5 bills of each type.

Solution: Suppose that a cash box has seven compartments, one to hold each type of bill. These compartments are separated by 6 dividers. The choice of 5 bills corresponds to placing 5 markers in the compartments holding different types of bills.

The number of ways to select 5 bills corresponds to the number of ways to arrange 6 bars and 5 stars in a row with a total of 11 positions. This corresponds to the number of unordered selections of 5 objects from a set of 11 objects, which can be done in $C(11, 5) = \dfrac{11!}{5!6!} = 462$ ways.

2 THEOREM 2 There are $C(n + r - 1, r) = C(n + r - 1, n - 1)$ r-combinations from a set with n elements when repetition of elements is allowed.

▼ Example 4

Suppose that a cookie shop has four different kinds of cookies. How many different ways can six cookies be chosen? Assume that only the type of cookie, and not the individual cookies or the order in which they are chosen, matters.

Solution: The number of ways to choose six cookies is the number of 6-combinations of a set with four elements. From Theorem 2 this equals $C(4 + 6 - 1, 6) = C(9, 6) = 84$ ways to choose the size cookies.

TABLE 1 Combinations and Permutations With and Without Repetition.

Type	*Repetition Allowed?*	*Formula*
r-permutations	No	$\dfrac{n!}{(n-r)!}$
r-combinations	No	$\dfrac{n!}{r!\,(n-r)!}$
r-permutations	Yes	n^r
r-combinations	Yes	$\dfrac{(n+r-1)!}{r!\,(n-1)!}$

Permutations with Indistinguishable Objects

3 THEOREM 3 The number of different permutations of n objects, where there are n_1 indistinguishable objects of type 1, n_2 indistinguiushable objects of type 2, ..., and n_k indistinguishable objects of type k, is $\dfrac{n!}{n_1!n_2!\cdots n_k!}$.

▼ Example 7

How many different strings can be made by reordering the letters of the word $SUCCESS$?

Solution: This word contains three Ss, two Cs, one U, and one E. Note that the three Ss can be placed among hte secen positions in $C(7.3)$ differen ways, leaving four positions free. Then the two Cs can be placed in $C(4, 2)$, leaving two free positions. The U can be placed in $C(2, 1)$ ways, leaving just one position free. Hence E can be placed in $C(1, 1)$ way. Consequently, from the product rule, the number of different strings that can be made is

$$C(7, 3)C(4, 2)C(2, 1)C(1, 1) = \frac{7!}{3!4!} \cdot \frac{4!}{2!2!} \cdot \frac{2!}{1!1!} \cdot \frac{1!}{1!0!} = \frac{7!}{3!2!1!1!} = 420.$$

Distributing Objects into Boxes

- **distinguishable (labeled)**: different from each other
- **indistinguishable (unlabeled)**: identical

Distunguishable Objects & Distinguishable Boxes

4 **THEOREM 4** The number of ways to distribute n distinguishable objects into k distinguishable boxes so that $n_1$objects are placed into box $i, i = 1, 2, \ldots, k$, equals $\frac{n!}{n_1!n_2! \cdots n_k!}$.

▼ Example 8

How many ways are there to distribute hands of 5 cards to each of four players from the standard deck of 52 cards?

Solution: Note that the first player can be dealt 5 cards in $C(52, 5)$ ways. The second player can be dealt 5 cards in $C(47, 5)$ ways, because only 47 cards are left. The third player can be dealt 5 cards in $C(42, 5)$ ways. Finally, the fourth player can be dealth 5 cards in $C(37, 5)$ ways. Hence, the total number of ways to deal four players 5 cards each is $C(52, 5)C(47, 5)C(42, 5)C(37, 5) = \frac{52!}{47!5!} \cdot \frac{47!}{42!5!} \cdot \frac{42!}{37!5!} \cdot \frac{37!}{32!5!} = \frac{52!}{5!5!5!5!32!}$.

Indistinguishable Objects & Distinguishable Boxes

▼ Example 9

How many ways are there to place 10 indistinguishable balls into 8 distinguishable bins?

The number of ways to place 10 indistinguishable balls into 8 bins equals the number of 10-combinations from a set with 8 elements when repetition is allowed. Consequently, there are $C(8 + 10 - 1, 10) = C(17, 10) = \frac{17!}{10!7!} = 19,448$.

$\Rightarrow$There are $C(n + r - 1, n - 1)$ ways to place r indistinuishable objects into n distinguishable boxes.

Distinguishable Objects & Indistinguishable Boxes

▼ Example 10

How many ways are there to put 4 different employees into 3 indistinguisable offices, when each office can contain any number of employees?

Solution: We will solve this problem by enumerating all the ways these employees can be placed into the offices. We represent the 4 employees by A, B, C, and D.

We can put all 4 employees into one office in exactly one way, represented by {{A, B, C, D}}; 3 employees into 1 office and the fourth employee into a different office in exactly 4 ways, {{A, B, C}, {D}}, {{A, B, D}, {C}}, {{A, C, D}, {B}}, and {{B, C, D}, {A}}; 2 employees into 1 office and two into a second office in exactly 3 ways, {{A, B}, {C, D}}, {{A, C}, {B, D}}, and {{A, D}, {B, C}}; finally, 2 employees into 1 office, and one each into each of the remaining 2 offices in 6 ways, {{A, B}, {C}, {D}}, {{A, C}, {B}, {D}}, {{A, D}, {B}, {C}}, {{B, C}, {A}, {D}}, {{B, D}, {A}, {C}}, and {{C, D}, {A}, {B}}. The total posiibilityes are 14 ways.

Another way to look at this problem is to look at the number of offices into which we put employees. Note that there are 6 ways to put four different employees into 3 indistinguishable offices so that no office is empty, 7 ways to put 4 different employees into 2 indistinguishable offices so that no office is empty, and 1 way to put 4 employees into one office so that it is not empty.

Stirling number of the second kind: Let $S(n, j)$ denote the number of ways to distribute n distinguishable objects into j indistinguishable boxes so that no box is empty.

Indistinguishable Objects & Indistinguishable Boxes

▼ Example 11

How many ways are there to pack 6 copies of the same book into 4 identical boxes, where a box can contain as many as 6 books?

Solution: We will enumerate all ways to pack the books. For each way to pack the books, we will list the number of books in the box with the largest number of books, followed by the numbers of books in each box containing at least one book, in order of decreasing number of books in a box. The ways we can peack the books are [6], [5, 1], [4, 2], [4, 1, 1], [3, 3], [3, 2, 1], [3, 1, 1, 1], [2, 2, 2], [2, 2, 1, 1]. (I.E. [4, 1, 1] indicates that 1 box contains 4 books, a second box contains a gingle book, and a third box contains a single book (and the fourth box is empty)). We conclude that there are 9 allowable ways to pack the books.

Ch6.6 Generating Permutations and Combinations

Generating Permutations

▼ Example 2

What is the next permutation in lexicographic order after 362541?

Solution: The last pair of integers a_j and a_{j+1} where $a_j < a_{j+1}$ is $a_3 = 2$ and $a_4 = 5$. The least integer to the right of 2 that is greater than 2 in the permutation is $a_5 = 4$. Hence, 4 is placed in the third position. Then the integers 2, 5, and 1 are placed in order in the last three positions, giving 125 as the last three positions of the permutation. Hence, the next permutation is 364125.

My Solution: The last permutation is 362514 by eye, it is clear that 362541 is the largest starting with "362", so the next permutation starts with "364", as 4 is the next smallest number after "36". 1, 2, and 5 are left, and 125 is the smallest form of the three numbers, so the next permutation is 364125.

Generating Combinations

▼ Example 5

Find the next larger 4-combination of the set {1, 2, 3, 4, 5, 6} after {1, 2, 5, 6}.

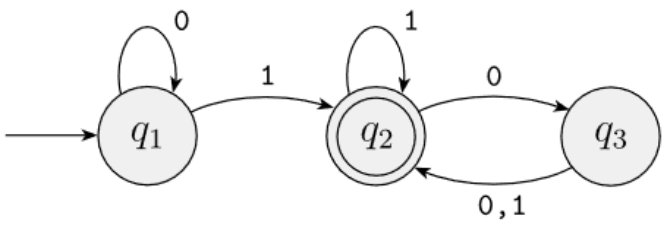
Solution: The last term among the terms a_i with $a_1 = 1, a_2 = 2, a_3 = 5$, and $a_4 = 6$ such that $a_i \neq 6 - 4 + i$ is $a_2 = 2$. To obtain the next larger 4-combination, increment a_2 by 1 to obtain $a_2 = 3$. Then set $a_3 = 3 + 1 = 4$ and $a_4 = 3 + 2 = 5$. Hence, the next larger 4-combination is {1, 3, 4, 5}.

My solution: It is clear that {1, 2, 5, 6} is the current largest set starting with {1, 2, x, x}, so the next largest 4-combination must start with {1, 3, x, x}, as 3 is the next largest number in the set after 1 and 2. After knowing the first two numbers, chooes the next two smallet number from the remaining set, which is 4 and 5 from {4, 5, 6}, putting them in order makes {1, 3, 4, 5} as the answer.

Ch 1.1 Finite Automata (S)

Markov chains ?

State diagram



M_1

- 3 states, labeled q_1, q_2, q_3, output *accept/reject*
- **start state**: q_1 (indicated by the arrow from nowhere)
- **accept state/final states**: q_2 (double circle)
- **transitions**: arrows

▼ Ex. 1101

1. start in state q_1
2. Read 1, follow transition from q_1 to q_2
3. Read 1, follow transition from q_2 to q_2
4. Read 0, follow transition from q_2 to q_3
5. Read 1, follow transition from q_3 to q_2
6. *Accept* because M_1 is in an accept state q_2 at the end

Formal Definition of a Finite Automaton

A *finite automaton* is a 5-tuple $(Q, \sum, \delta, q_0, F)$, where
1. Q is a finite set called the **states**,
2. $\sum$ is a finite set called the **alphabet**,
3. $\delta : Q \times \sum \longrightarrow Q$ is the **transition function**,
4. $q_0 \in Q$ is the **start state**, and
5. $F \subseteq Q$ is the **set of accept states**.

▼ Example M_1

We can describe M_1 formally by writing $M_1 = (Q, \sum, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\sum = \{0, 1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

4. q_1 is the start state, and
5. $F = \{q_2\}$.

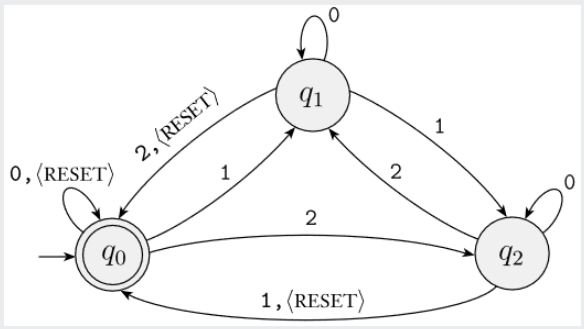
$A = \{w | w$ contains at least one 1 and an even number of 0s follow the last $1\}$.

$L(M_1) = A$, or equivalently, M_1 recognizes A.

> Let A be the set of all strings M accepts, A is the language of machine M and write $L(M) = A$. We say that M recognizes A.

Examples of Finite Automata

▼ Example 1.13: Modulo 3



*view <RESET> as a single symbol

$\sum = \{<\text{RESET}>, 0, 1, 2\}$

This machine is running count of the sum of the numerical input symbols it reads, modulo 3.

▼ Example 1.15: no diagram but description

Consider a generalization of Example 1.13, using the same four-symbol alphabet $\sum$. For each $i \geq 1$ let A_i be the language of all strings where the sum of the numbers is a multiple of i, except that the sum is reset to 0 whenever the symbol <RESET> appears. For each A_i we give a finite automaton $B_i = (Q_i, \sum, \delta_i, q_0, \{q_0\})$, where Q_i is the set of i states $\{q_0, q_1, q_2, \ldots, q_{i-1}\}$, and we design the transition function δ_i so that for each j, if B_i is in q_j, the running sum is j, modulo i. For each q_j let

$\delta_i(q_j, 0) = q_j$,

$\delta_i(q_j, 1) = q_k$, where $k = j + 1 \mod i$,

$\delta_i(q_j, 2) = q_k$, where $k = j + 2 \mod i$, and

$\delta_i(q_j, <\text{RESET}>) = q_0$,

Formal Definition of Computation

Let $M = (Q, \sum, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \ldots w_n$ be a string where each w_i is a member of the alphabet $\sum$. Then M **accepts** w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and
3. $r_n \in F$.

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

Designing Finite Automata

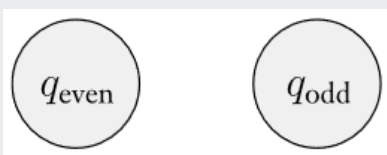
- put *yourself* in the place of the machine
- only remember certain crucial input

▼ All strings with an odd number of 1s.

$\sum = \{0, 1\}$

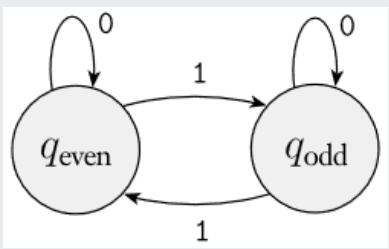
Possibilities

1. even so far, and
2. odd so far



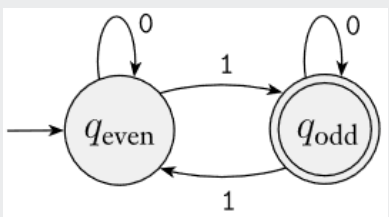
▼

Assign transitions (seeing how to go from one possibility to another)



set start state → the possibility having seem 0 symbols so far (ϵ) → q_{even} ∵ 0 is even

set accept states → q_{odd}



The Regular Operations

Let A and B be languages. We define the regular operations **union, concatenation,** and **star** as follows:

- **Union**: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$
- **Concatenation:** $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$
- **Star**: $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$

▼ Example 1.24

Let the alphabet $\sum$ be the standard 26 letters $\{a, b, \dots, z\}$. If $A = \{\text{good, bad}\}$ and $B = \{\text{boy, girl}\}$, then

$A \cup B = \{\text{good, bad, boy, girl}\}$,

$A \circ B = \{\text{goodboy, goodgirl, badboy, badgirl}\}$, and

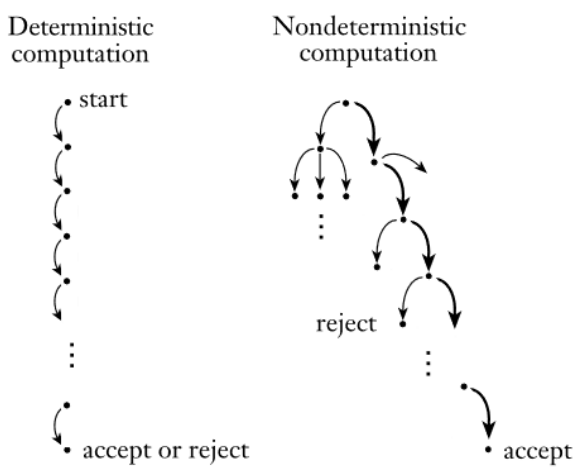
$A^* = \{\epsilon, \text{good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, goodbadgood, goodbadbad, } \dots\}$

The class of regular languages is closed under the union operation.
In other words, if A_1 and A_2 are regular languages, so is $A_1 \cup A_2$.

The class of regular languages is closed under the concatenation operation.
In other words, if A_1 and A_2 are regular languages, so is $A_1 \circ A_2$.

Ch 1.2 Nondeterminism (S)

- **deterministic** computation: when the machine is in a given state and reads the next input symbol, we know what the next state will be.
- **nondeterministic** machine: several choices may exist for the next state
- **DFA**: deterministic finite automaton; **NFA**: nondeterministic finite automaton.



▼ How NFA compute?

It follows all posibilities in parallel, splits into multiple copies of itself.

If there are subsequent choices, the machine splits again.

If the next input doesnot appear on any arrow existing, the copy dies.

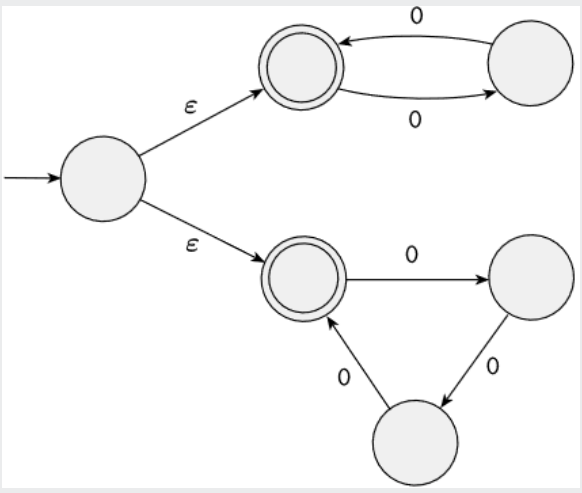
If *any one* of these copies ends up in the accept state, the NFA accepts the input string.

▼ How ϵ works?

If a state with ϵ exists, without reading any input, the machine splits into multiple copies, one stays at the current state, one follow the arrow with the ϵ-label.

▼ Example 1.33

The following NFA N_3 has an input alphabet $\{0\}$ consisting of a single symbol. An alphabet containing only one symbol is called a ***unary alphabet***.



This machine demonstrates the convenience of having ϵ arrows. It accepts all strings of the form 0^k where k is a multiple of 2 or 3. (The subscript denotes repetition, not numerical exponentiation). Ex. N^3 accepts the strings $\epsilon, 00, 000, 0000, 000000$, but not $0, 00000$.

Formal Definition of a NFA

A *nondeterministic finite automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where
1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta : Q \times \Sigma_\epsilon \longrightarrow P(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

Equivalence of NFAs and DFAs

- Two machines are **equivalent** if they recognise the same language.

Theorem Every NFA has an equivalent DFA.

Corollary A language is regular *iff* some NFA recognises it.

Closure Under the Regular Operations

Theorem The class of regular languages is closed under the *union* operation.

Theorem The class of regular languages is closed under the *concatenation* operation.

Theorem The class of regular languages is closed under the *star* operation.

Ch 1.3 Regular Expressions (S)

💡 Say that R is a **regular expression** if R is
1. a for some a in the alphabet Σ,
2. ϵ,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

▼ *Remark 1*

In items 1 & 2, the regular expressions a and ϵ represent the languages $\{a\}$ and $\{\epsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

▼ *Remark 2*

The expression ϵ represents the language containing a single string — namely, the empty string — whereas $\emptyset$ represents the language that doesn't contain any strings.

❓ Example 1.53

Equivalence with Finite Automata

Any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa.

 **Theorem** A language is regular *iff* some regular expression describes it.

 **Lemma** If a language is described by a regular expression, then it is regular.

Example 1.58 **Convert regular expression to an NFA**

 **Lemma** If a language is regular, then it is described by a regular expression.

💡 A **generalized nondeterministic finite automaton** is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where
1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta : (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{start}\}) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

Ch 1.4 Nonregular languages (S)

How to prove certain languages can't be recognized by any finite automaton.

The Pumping Lemma for Regular Languages

Pumping Lemma
If A is a regular 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 three pieces, $s = xyz$, satisfying the following conditions:
1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Remark: $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ϵ.

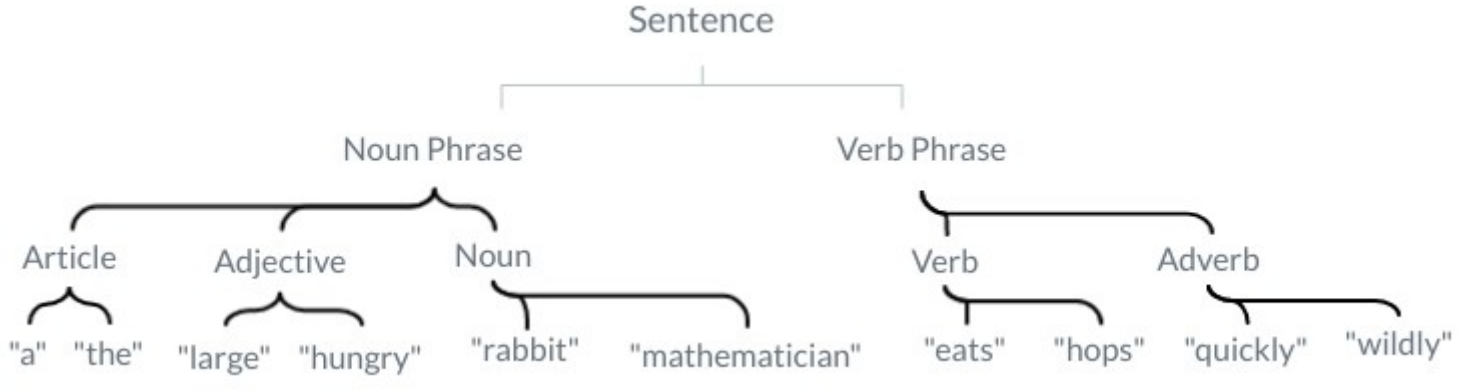
Skipped proof ideas and proof, examples.

Ch 13 Modeling Computation (R)

Structures used in models of computation: **grammar, finite-state machines,** and **Turing Machines**.

Ch 13.1 Languages and Grammars

- **Syntax** is the form, **semantics** is the meaning.



sentence
noun phrase verb phrase
article adjective noun verb phrase
article adjective noun verb adverb
the adjective noun verb adverb
the large noun verb adverb
the large rabbit verb adverb
the large rabbit hops adverb
the large rabbit hops quickly

A *vocabulary* (or *alphabet*) V is a finite, nonempty set of elements called *symbols*. A *word* (or *sentence*) over V is a string of finite length of elements of V. The *empty string* or *null string*, denoted by λ (and sometimes by ϵ), is the string containing no symbols. The set of all words over V is denoted by V^*. A *language over* V is a subset of V^*.

- λ, the empty string, is different from $\emptyset$, the empty set. $\{\lambda\}$ is the set containing exactly one string.
- V is the set of symbols used to derive members of the language
- **Terminal** are elements of the vocabulary that cannot be replaced by other sumbols, T.
- **Nonterminal** are elements otherwise, N.

A *phrase-structure grammar* $G = (V, T, S, P)$ consists of a vocabulary V, a subset T of V consisting of terminal symbols, a start symbol S from V, and a finite set of productions P. The set $V - T$ is denoted N. Elements of N are called *nonterminal symbols*. Every production in P must contain at least one nonterminal on its left side.

Let $G = (V, T, S, P)$ be a phrase-structure grammar. Let $w_0 = lz_0r$ (concatenation of l, z_0, r) and $w_1 = lz_1r$ be strings over V. If $z_0 \to z_1$ is a production of G, we say that w_1 is *directly derivable* from w_0 and we write $w_0 \Rightarrow w_1$. If $w_0, w_1, \ldots, w_n$ are strings over V such that $w_0 \Rightarrow w_1, w_1 \Rightarrow w_2, \ldots, w_{n-1} \Rightarrow w_n$, then we say that w_n is *derivable from* w_0, and we write $w_0 \overset{*}{\Rightarrow} w_n$. The sequence of steps used to obtain w_n from w_0 is called a *derivation*.

Let $G = (V, T, S, P)$ be a phrase-structure grammar. The *language generated by* G (or the *language of* G), denoted by $L(G)$, is the set of all strings of terminals that are derivable from the starting state S. In other words, $L(G) = \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$.

TABLE 1 Types of Grammars.	
Type	*Restrictions on Productions* $w_1 \to w_2$
0	No restrictions
1	$w_1 = lAr$ and $w_2 = lwr$, where $A \in N$, $l, r, w \in (N \cup T)^*$ and $w \neq \lambda$; or $w_1 = S$ and $w_2 = \lambda$ as long as S is not on the right-hand side of another production
2	$w_1 = A$, where A is a nonterminal symbol
3	$w_1 = A$ and $w_2 = aB$ or $w_2 = a$, where $A \in N$, $B \in N$, and $a \in T$; or $w_1 = S$ and $w_2 = \lambda$

- **type 1** grammar are called **context-sensitive**
- **type 2** grammars are called **context-free grammars**
- **type 3** grammars are called **regular grammars**

▼ Example 12 **Top-down parsing & Bottom-up parsing**

Determine whether the word *cbab* belongs to the language generated by the grammar $G = (V, T, S, P)$, where $V = \{a, b, c, A, B, C, S\}, T = \{a, b, c\}$, S is the starting symbol, and the productions are

$S \to AB$

$A \to Ca$

$B \to Ba$

$B \to Cb$

$B \to b$

$C \to cb$

$C \to b$

Solution: (Top-down approach)

$S \Rightarrow AB \Rightarrow CaB \Rightarrow cbaB \Rightarrow cbab$

Solution: (Bottom-up approach)

$cbab \Leftarrow Cab \Leftarrow Ab \Leftarrow AB \Leftarrow S$

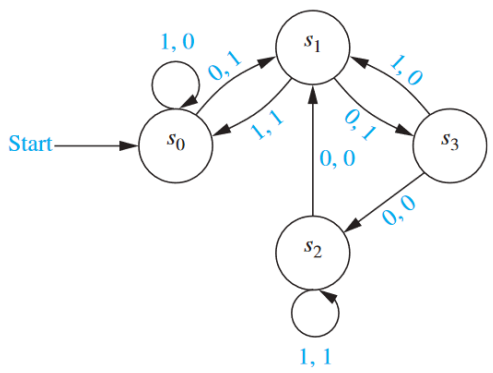
Ch 13.2 Finite-State Machines with Output

A *finite-state machine* $M = (S, I, O, f, g, s_0)$ consists of a finite set S of *states*, a finite *input alphabet* I, a finite *output alphabet* O, a *transition function* f that assigns to each state and input pair a new state, an *output function* g that assigns to each state and input pair an output, and an *initial state* s_0.

TABLE 2

	f		g	
	Input		*Input*	
State	0	1	0	1
s_0	s_1	s_0	1	0
s_1	s_3	s_0	1	1
s_2	s_1	s_2	0	1
s_3	s_2	s_1	0	0

Example of **state table**



Example of **state diagram**

Let $M = (S, I, O, f, g, s_0)$ be a finite-state machine and $L \subseteq I^*$. We say that M *recognizes* (or *accepts*) L if an input string x belongs to L *iff* the last output bit produced by M when given x as input is a 1.

Ch 13.3 Finite-State Machines with No Output

Suppose that A and B are subsets of V^*, where V is a vocabulary. The *concatenation* of A and B, denoted by AB, is the set of all strings of the form xy, where x is a string in A and y is a string in B.

▼ Example 1

Let $A = \{0, 11\}$ and $B = \{1, 10, 110\}$. Find AB and BA.

Solution: The set AB contains every concatenation of a string in A and a string in B. Hence, $AB = \{01, 010, 0110, 111, 1110, 11110\}$. The set BA contains every concatenation of a string in B and a string in A. Hence, $BA = \{10, 111, 100, 1011, 1100, 11011\}$.

Suppose that A is a subset of V^*. Then the *Kleene closure* of A, denoted by A^*, is the set consisting of concatenation of arbitrarily many strings from A. That is, $A^* = \bigcup_{k=0}^{\infty} A^k$.

▼ Example 3

What are the Kleene closures of the sets $A = \{0\}$, $B = \{0, 1\}$, and $C = \{11\}$?

Solution:

$A^* = \{0^n \mid n = 0, 1, 2, \dots\}$.

$B^* = V^*$

$C^* = \{1^{2n} \mid n = 0, 1, 2, \dots\}$

A *finite-state automaton* $M = (S, I, f, s_0, F)$ consists of a finite set S of *states*, a finite *input alphabet* I, a *transition function* f that assigns a next state to every pair of state and input (so that $f : S \times I \to S$), an *initial* or *start state* s_0, and a subset F of S consisting of *final* (or *accepting states*).

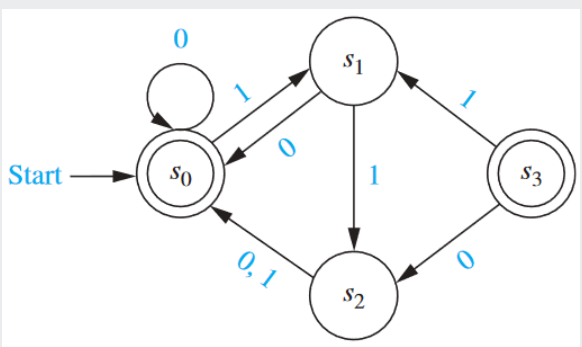
▼ Example 4

Construct the state diagram for the finite-state automaton $M = \{S, I, f, s_0, F\}$, where $S = \{s_0, s_1, s_2, s_3\}$, $I = \{0, 1\}$, $F = \{s_0, s_3\}$, and the transition function f is given.

TABLE 1

	f	
	Input	
State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_0
s_3	s_2	s_1

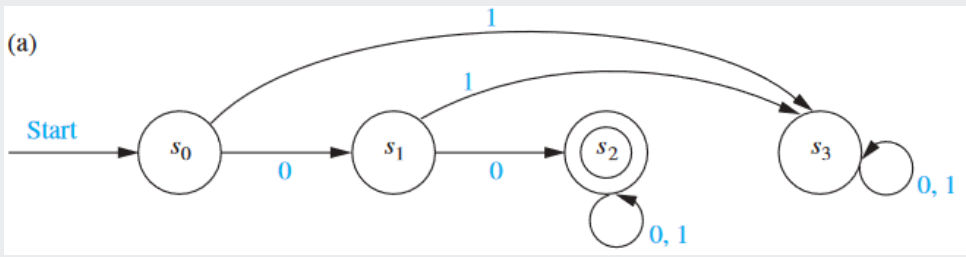
Solution:



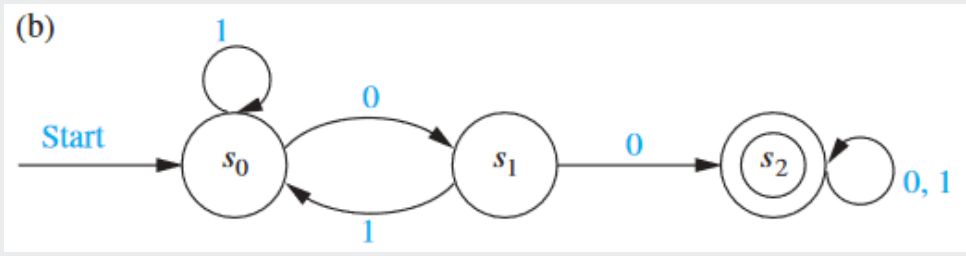
A string x is said to be *recognized* or *accepted* by machine $M = (S, I, f, s_0, F)$ if it takes the initial state s_0 to a final state, that is, $f(s_0, x)$ is a state in F. The *language recognized* or *accepted* by the machine M, denoted by $L(M)$, is the set of all strings that are recognized by M. Two finite-state automata are called *equivalent* if they recognize the same language.

▼ Example 6

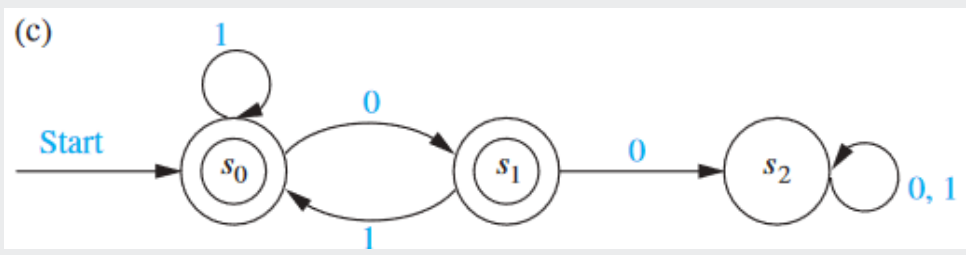
1. The set of bit strings that begin with two 0s



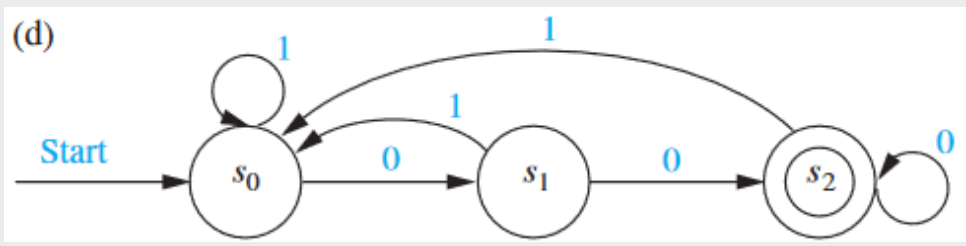
2. The set of bit strings that contain two consecutive 0s



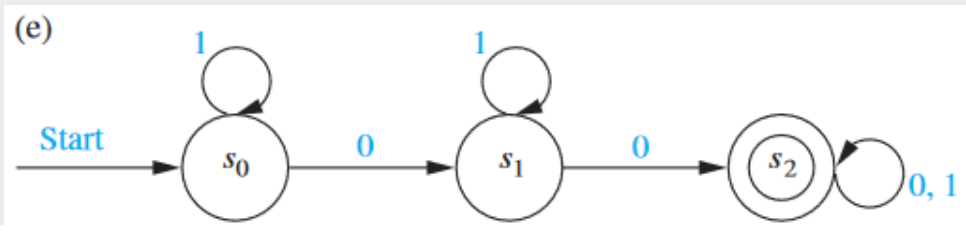
3. The set of bit strings that do not contain two consecutive 0s



4. The set of bit strings that end with two 0s

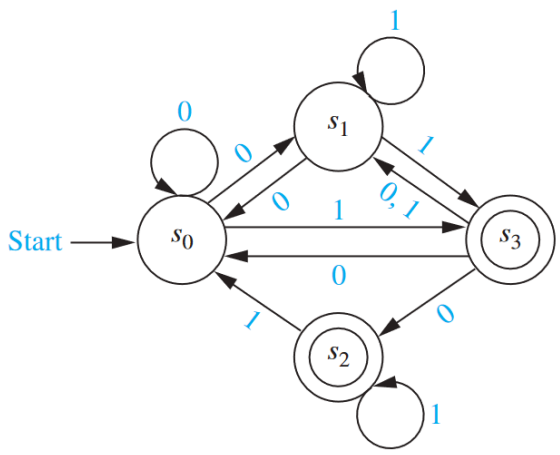


5. The set of bit strings that contain at least two 0s



A *nondeterministic finite-state automaton* $M = (S, I, f, s_0, F)$ consists of a finite set S of *states*, a input alphabet I, a transition function f that assigns a set of states to each pair of state and input (so that $f : S \times I \to P(S)$), an starting state s_0, and a subset F of S consisting of the final states.

TABLE 2		
	f	
	Input	
State	**0**	**1**
s_0	s_0, s_1	s_3
s_1	s_0	s_1, s_3
s_2		s_0, s_2
s_3	s_0, s_1, s_2	s_1



If the language L is recognized by a nondeterministic finite-state automaton M_0, then L is also recognized by a deterministic finite-state automaton M_1.

Ch 13.4 Language Recognition

The *regular expressions* over a set I are defined recursively by:
the symbol $\emptyset$ is a regular expression;
the symbol λ is a regular expression;
the symbol x is a regular expression whenever $x \in I$;
the symbols (AB), $(A \cup B)$, A^* are regular expressions whenever A and B are regular expressions.

KLEENE'S THEOREM A set is regular *iff* it is recognized by a finite-state automaton.

A set is generated by a regular grammar *iff* it is a regular set.

Ch 3.1 Algorithms (R)

Algorithms

An *algorithm* is a finite sequence of precise instructions for performing a computation or for solving a problem.

Properties of Algorithms

Input: An algorithm has input values from a specified set

Output: From each set of input values an algorithm produces output values from a specified set.

The output values are the solution to the problem.

Definiteness: The steps of an algorithm must be defined precisely.

Correctness: An algorithm should produce the correct output values for each set of input values.

Finiteness: An algorithm should produce the desired output after a finite (but perhaps large) number of steps for any input in the set.

Effectiveness: It must be possible to perform each step of an algorithm exactly and in a finite amount of time.

Generality: The procedure should be applicable for all problems of the desired form, not just for a particular set of input values.

Searching Algorithms

The Linear Search (Sequential Search)

Start comparing with the first term, see if it matches, and go on.

ALGORITHM 2 The Linear Search Algorithm.

procedure *linear search*(x: integer, $a_1, a_2, \ldots, a_n$: distinct integers)
$i := 1$
while ($i \leq n$ and $x \neq a_i$)
    $i := i + 1$
if $i \leq n$ **then** *location* $:= i$
else *location* $:= 0$
return *location*{*location* is the subscript of the term that equals x, or is 0 if x is not found}

The Binary Search

Split into two lists, compare the largest term in the first list, eliminate one list, repeat until one list with one term is left.

ALGORITHM 3 The Binary Search Algorithm.

procedure *binary search* (x: integer, $a_1, a_2, \ldots, a_n$: increasing integers)
$i := 1$ {i is left endpoint of search interval}
$j := n$ {j is right endpoint of search interval}
while $i < j$
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ **then** $i := m + 1$
    else $j := m$
if $x = a_i$ **then** *location* $:= i$
else *location* $:= 0$
return *location*{*location* is the subscript i of the term a_i equal to x, or 0 if x is not found}

Sorting

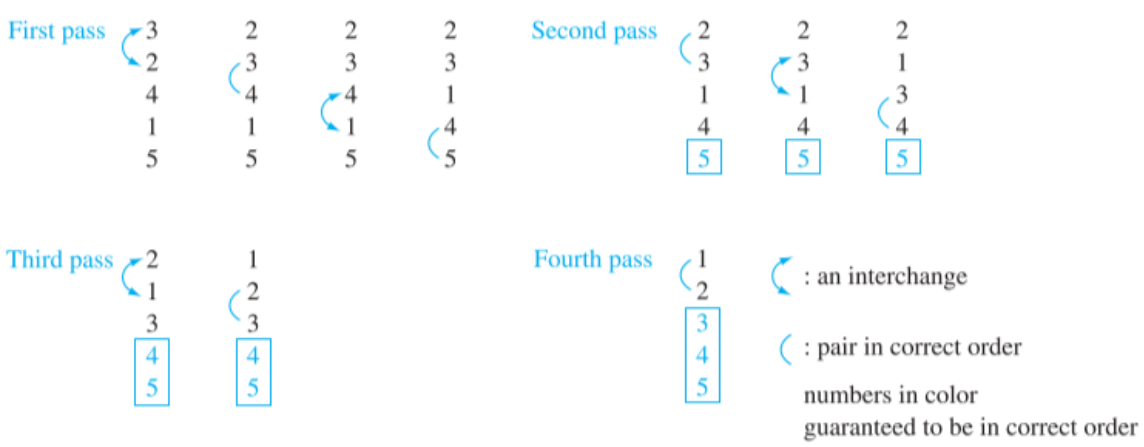
Bubble Sort

One of the simplest but not one of the most efficient

Successively comparing adjacent elements, interchange them if in wrong order.

ALGORITHM 4 The Bubble Sort.

procedure *bubblesort*($a_1, \ldots, a_n$: real numbers with $n \geq 2$)
for $i := 1$ **to** $n - 1$
  for $j := 1$ **to** $n - i$
    if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}
{$a_1, \ldots, a_n$ is in increasing order}



Insertion Sort

A simple sorting algorithm, but usually not the most efficient.

Begins with the second element, compare the second to the first, if exceed, insert before the first element if does not exceed, vice versa. And on with the third... elements.

ALGORITHM 5 The Insertion Sort.

procedure *insertion sort*($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)
for $j := 2$ **to** n
  $i := 1$
  while $a_j > a_i$
    $i := i + 1$
  $m := a_j$
  for $k := 0$ **to** $j - i - 1$
    $a_{j-k} := a_{j-k-1}$
  $a_i := m$
{$a_1, \ldots, a_n$ is in increasing order}

String Matching

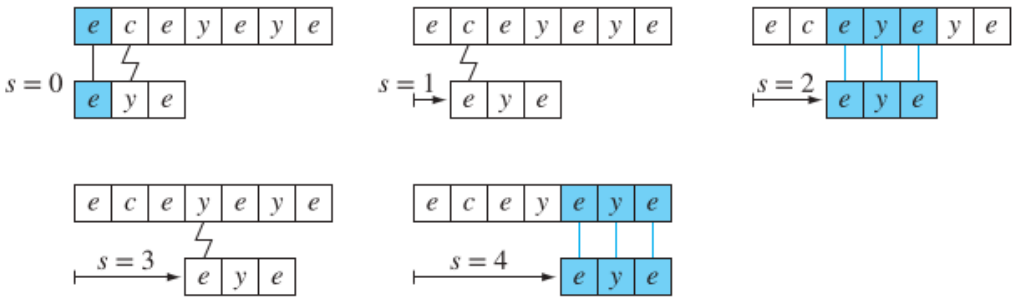
→ Finding where a pattern occurs in a text string

Ex. Find whether 101 is in 11001011, whether CAG is in CATCACAGAGA.

Naive String Matcher

ALGORITHM 6 Naive String Matcher.

procedure *string match* (n, m: positive integers, $m \leq n$, $t_1, t_2, \ldots, t_n, p_1, p_2, \ldots, p_m$: characters)
  for $s := 0$ **to** $n - m$
    $j := 1$
    while ($j \leq m$ and $t_{s+j} = p_j$)
      $j := j + 1$
    if $j > m$ **then print** "s is a valid shift"



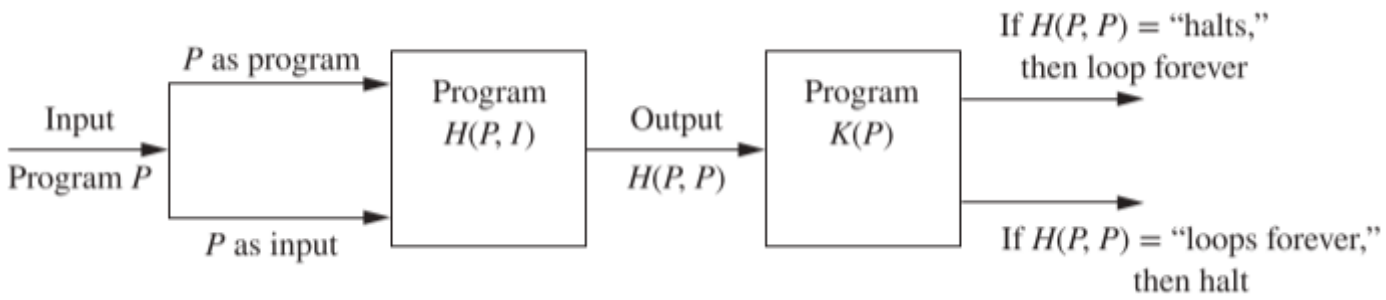
Greedy Algorithms

→ solve optimization problems

→ Algorithms that make what seems to be the "best" choice at each step

→ We call the algorithm "greedy" whether or not it finds an optimal solution

The Halting Problem



Can't just give an input to test, because

if it halts, we have our answers

but if it is still running after any fixed length of time, we do not know whether it will never halt or we just did not wait long enough for it to terminate.

Appendix 3 Pseudocode

```
# precedure statements
procedure maximum(L: list of integers)

# assignments
variable := expression
max := a
x := largest integer in the list L
interchange a and b

# comments
{ x is the largest element in L }

# conditional
if condition then statement

if condition then
  block of statements

if condition then
  statement 1
  statement 2
  statement 3
  .
  .
  .
  statement n

if condition then statement 1
else statement 2

if condition 1 then statement 1
else if condition 2 then statement 2
else if condition 3 then statement 3
  .
  .
  .
else if condition n then statement n
else statement n + 1

# loop
for variable := initla value to final value
  (block of) statement(s)

sum := 0
for i := 1 to n
  sum := sum + i

for all elements with a certain property

while condition
  (block of) statement(s)

sum := 0
while n > 0
  sum := sum + n
  n := n - 1

# precedures in other precedures
max(L)

# return statements
return x
return f(n-1)
```

Ch 5.4 Recursive Algorithms (R)

🗣️ An algorithm is called *recursive* if it solves a problem by reducing it to an instance of the same problem with smaller input.

Can use either mathematical induction or strong induction to prove its correctness.

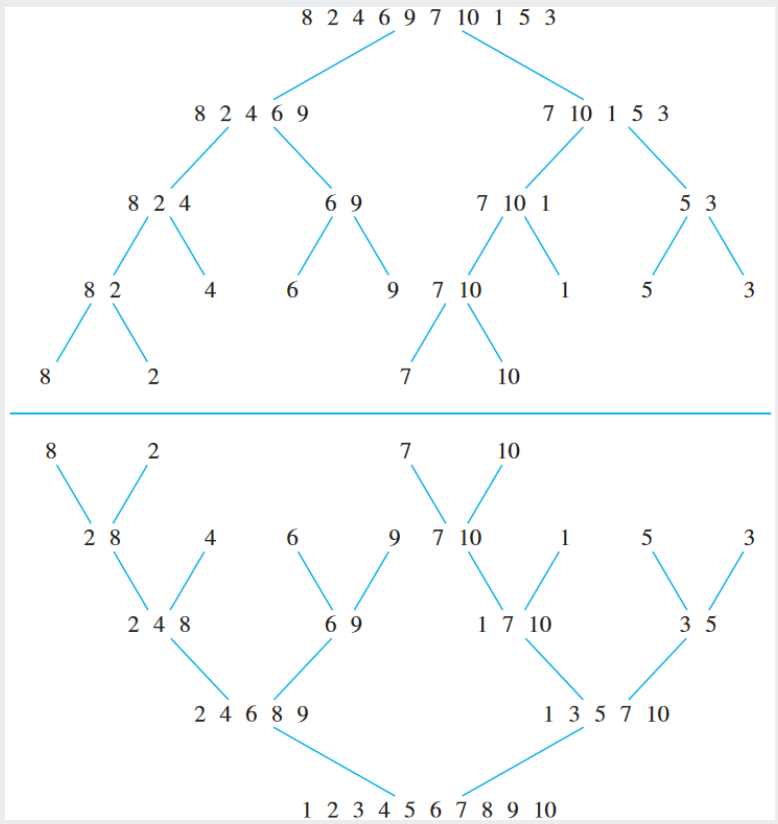
Iterative: start with the value of the function at one or more integers, the base cases, and successively apply the recursive definition to find the values of the function at successive larger integers.

Often use for the evaluation of a recursively defined sequence that requires much less computation.

▼ Example 9

Use the merge sort to put the terms of the list 8, 2, 4, 6, 9, 7, 10, 1, 5, 3 in increasing order.

1. Split list into two successively until it becomes individual elements
2. Merge by pairs successively in increasing order



ALGORITHM 9 A Recursive Merge Sort.

procedure *mergesort*($L = a_1, \ldots, a_n$)
if $n > 1$ **then**
    $m := \lfloor n/2 \rfloor$
    $L_1 := a_1, a_2, \ldots, a_m$
    $L_2 := a_{m+1}, a_{m+2}, \ldots, a_n$
    $L := merge(mergesort(L_1), mergesort(L_2))$
{L is now sorted into elements in nondecreasing order}

💡 Two sorted lists with m elements and n elements can be merged into a sorted list using no more than $m + n - 1$ comparisons.

🗣️ The number of comparisons needed to merge sort a list with n elements is $O(n \log n)$.

Ch 9 Sorting (DS)

Internal Sort: Insertion Sort, Bubble Sort, Quick Sort, and Heap Sort. (RAM)

External Sort: Merge Sort.(disks, taps, ROMs)

Data compositions: files of records containing keys

File: collection of records

Record: collection of fields

Field: collection of characters

Key: uniquely identifies a record in a file, used to control the sort

Selection Sort

Select successive elements in ascending order, place them into their proper order.

Find the smallest, exchange it with the first, find the second smallest, exchange it with the second, etc.

```
void selection_sort(int array[], int n){
  int i, j, min, tmp;
  for (i = 0; i < n; i++){
    min = i;
    for (j = 0; j < n; j++){
      if (array[j] < array[min]) min = j;
    }
    tmp = array[min];
    array[min] = array[i];
    array[i] = tmp;
  }
}
```

Insertion Sort

Take elements one by one, insert them by proper position among these already taken and sorted in a new collection, repeat until done.

```
void insertion_sort(int array[], int n){
  int i, j, tmp;
  for (i = 0; i < n; i++){
    tmp = array[i];
    for (j = 0; j < n; j++){
      array[j] = array[j-1];
    }
    array[j] = tmp;
  }
}
```

Bubble Sort

Compare two elements in the neighbor, exchange them if not in proper order. The greatest element moves to the most right, repeat except the most right element until done.

```
void bubble_sort(int array[], int n){
  int i, j, tmp;
  for (i = n; i < 0; i--){
    for (j = 1; j < i; j++){
      if (array[j-1] > array[j]){
        tmp = array[j-1];
        array[j-1] = array[j];
        array[j] = tmp;
      }
    }
  }
}
```

Quick Sort

Divide and Conquer

Choose a pivotal (central) element, move elements smaller than that to the left and greater to the right. Repeat with smaller lists until done.

```
void quick_sort(int array[]. int left, int right){
  int i;
  if (right > left){
    i = partition(left, right);
    quick_sort(array, left, i-1);
    quick_sort(array, i+1, right);
  }
}
// partition(left, right) =>
// when the element array[i] is placed in its final place, all elements of array[left], ..., array[i-1] are smaller than or equal to arr
ay[i], and all elements of array[i+1], ..., array[right] are larger than or equal to array[i]. That is, array[j] <= array[i] for j < i a
nd array[j] >= array[i] for j > i


void quick_sort(int array[]. int left, int right){
  int v, i, j, tmp;
  if (right > left){
    v = array[right];
    i = left - 1;
    j = right;
    for (;;){
      while (array[++i] < v);
      while (array[--j] > v);
      if (i >= j) break;
      tmp = array[i];
      array[i] = array[j];
      array[j] = tmp;
    }
    tmp = array[i];
    array[i] = array[right];
    array[right] = tmp;
    quick_sort(array, left, i-1);
    quick_sort(array, i+1, right);
  }
}
```

Heap Sort

```
void down_heap(int array[], int n, int x){
  int i, v;
  v = array[x];
  while (x <= n/2){
    i = x+x;
    if (i < n && array[i] < array[i+1]) i++;
    if (v >= array[i]) break;
    array[x] = array[i];
    x = i;
  }
  array[x] = v;
}


void heap_sort(int array[]. int n){
  int x, tmp;
  for (x = n/2; x >= 0; x--){
    down_heap(array, n, x);
  }
  while (n > 0){
    tmp = array[0];
    array[0] = array[n];
    array[n] = tmp;
    down_heap(array, --n, 0);
  }
}
```

Merge Sort

```
void merge_sort(int a[], int l, int r){
  int i, j, k, m;
  if (r > l){
    m = (r+l)/2;
    merge_sort(a, l, m);
    merge_sort(a, m+l, r);
```

```
    for (i = m+1; i > l; i--) b[i-1] = a[i-1];
    for (j = m; j < r; j++) b[r+m-j] = a[j+1];
    for (k = l; k <= r; k++)
      a[k] = (b[i] < b[j]) ? b[i++] : b[j--];
  }
}
```

Shell Sort

improvement of Insertion Sort

```
void shell_sort(int array[], int n){
  int i, j, k, tmp;
  for (k = 1; k < n/9; k = 3*k+1);
  for (; k > 0; k /= 3){
    for (i = k; i < n; i++){
      j = i;
      while (j >= k && array[j-k] > array[j]){
        tmp = array[j];
        array[j] = array[j-k];
        array[j-k] = tmp;
        j -= k;
      }
    }
  }
}
```

Radix Sort

Scans from left to find a key which starts with bit 1, scans from the right to find a key which starts with bit 0, then exchange, continue until done.

```
void radix_exchange(int array[], int l; int r, int b){
  int i, j, tmp;
  if (r > l && b >= 0){
    i = l;
    j = r;
    while (j != i){
      while (bits(array[i], b, 1) == 0 && i < j) i++;
      while (bits(array[j], b, 1) != 0 && j < i) j--;
      tmp = array[i];
      array[i] = array[j];
      array[j] = tmp;
    }
    if (bits(array[r], b, 1) == 0) j++;
    radix_exchange(array, l, j-1, b-1);
    radix_exchange(array, j, r, b-1);
  }
}
```

Algorithms	Performance	Comments
Selection Sort	N^2	good for small and partially sorted data
Insertion Sort	N^2	good for almost sorted data
Bubble Sort	N^2	good for $N < 100$
Quick Sort	$N \log N$	excellent
Heap Sort	$N \log N$	excellent
Merge Sort	$\log N$	good for external sort
Shell Sort	$N^{1.5}$	good for medium N
Radix Sort	$N \log N$	excellent