Computer Science for Visitants

Jalal Kawash

CPSC 203 Course Notes

Contents

Chapter 1
First Things First: Logic and Set Theory

Logic and sets are central to understanding Computer Science. Logic aims at helping us give formal unambiguous statements and the rules of logic provide us with a very important tool: *reasoning*. Mathematics is crucial for computers and logic is at the heart of Mathematics. Logic is the basis of designing computing devices and formulating computer programs.

Much of Computer Science is devoted to the study of objects and set theory comes in handy for the representation of objects and their relationships. A set is simply a collection of unordered objects and a relation is a set of ordered pairs. Graphs, trees, and finite state machines (which will be studied later) are founded on set theory.

This chapter serves as a required background for the coming chapters. It must be carefully studied before attempting subsequent chapters.

I – Propositional Logic

A *proposition* is a declarative sentence which must be *true* or *false*, but not both. True and false are called *truth values*. The area of logic that studies propositions is called *propositional logic (or calculus)*.

EXAMPLE 1 – Propositions

1. Calgary is the capital of Canada
2. British Columbia is a province of Canada
3. $6 + 5 < 3$
4. $6 + 5 > 3$

All of these statements are propositions; each has a truth value which is either true or false. For instance propositions 1 and 3 are false, but 2 and 4 are true. The following statements are not propositions:

5. Do you like CPSC 203?
6. Study hard for this course
7. $2x > 12$

Statement 7 is worth a little more explanation. It is not possible to determine the truth value of statement 7, if the value of x is unknown. The statement may be true (for instance when x is 10 or 11) and could be false as well (when x is 1 or 5). This statement can be converted to a proposition if the value of x is fixed. However, we will see later how to convert it to a proposition without fixing the value of the variable x, using *predicate logic*.

Compound propositions: *Compound* propositions can be built by combining one or more propositions, using logical operators. There are three major logical operators:

1. Negation, the *not* operator
2. Conjunction, the *and* operator
3. Disjunction, the *or* operator

EXAMPLE 2 – Compound propositions

The following are compound propositions:

1. It is raining today **AND** it is very warm
2. At 12:00 today, I will be eating **OR** I will be home
3. I will **NOT** be home at 6

If P is a proposition, then NOT P is also a proposition. NOT P is true when P is false and it is false when P is true. This is typically defined using a *truth table*, which simply displays all the combinations of truth values of propositions.

Truth table for negation

P	NOT P
F	T
T	F

If P and Q are propositions, then P AND Q is also a proposition. P AND Q is true when both P and Q are true, otherwise (at least one of P or Q is false) it is false.

Truth table for conjunction

P	Q	P AND Q
F	F	F
F	T	F
T	F	F
T	T	T

If P and Q are propositions, then P OR Q is also a proposition. P OR Q is false when both P and Q are false, otherwise (at least one of P or Q is true) it is true.

Truth table for disjunction

P	Q	P OR Q
F	F	F
F	T	T
T	F	T
T	T	T

The *or* operator is *inclusive*. P OR Q simply states that the compound proposition is true if P is true, Q is true, or both are true. Often, we use an *exclusive* version of *or* in our daily logic. The proposition: *Tonight, I will be at home or at school* does not admit inclusion. That is, if I am at home, then definitely I am not at school and vice-versa. The proposition *I will be at home* and the proposition *I will be at school* mutually exclude each other. In other words, both cannot be true at the same time. This gives rise to a fourth logic operator *exclusive or* (abbreviated XOR).

If P and Q are propositions, then P XOR Q is also a proposition. P XOR Q is true when exactly one of P or Q is true, otherwise it is false.

Truth table for exclusive or

P	Q	P XOR Q
F	F	F
F	T	T
T	F	T
T	T	F

Logical equivalence: Two compound propositions are said to be *logically equivalent*, if they have the same truth values in all possible cases. A truth table is a handy tool in showing logical equivalence.

EXAMPLE 3 – XOR

The XOR operation is redundant in the sense that it can be expressed using the three core operators: **not**, **and** , and **or**. The proposition P XOR Q is equivalent to (P OR Q) AND NOT (P AND Q) as is demonstrated by the following truth table. It can be verified from this table that the truth values in columns 3 and 7 are the same.

P	Q	P XOR Q	P OR Q	P AND Q	NOT (P AND Q)	(P OR Q) AND NOT (P AND Q)
F	F	F	F	F	T	F
F	T	T	T	F	T	T
T	F	T	T	F	T	T
T	T	F	T	T	F	F

EXAMPLE 4 – De Morgan's rules

1. The proposition NOT (A AND B) is equivalent to (NOT A) OR (NOT B)
2. The proposition NOT (A OR B) is equivalent to (NOT A) AND (NOT B)

Verification of these logical equivalences can be done using truth tables and is left as an exercise.

Implication: Often compound propositions can be constructed using an if-then structure. We have used this structure in defining compound propositions: *If P is a proposition, then NOT P is also a proposition* … This form is captured in logic using the *implication* operator, denoted P → Q and read "P implies Q" or "If P then Q".

If P and Q are propositions, then P → Q is also a proposition. P → Q is false when P is true but Q is false, otherwise it is true.

Truth table for implication

P	Q	P → Q
F	F	T
F	T	T
T	F	F
T	T	T

The definition of implication is intriguing. It states that you can start from a false premise and arrive at any conclusion (true or false), but starting from a true premise, you can only arrive at true conclusions.

EXAMPLE 5 – Implication

Consider the proposition: *If you have a Canadian passport, then you are a Canadian citizen*. If we let P = "*You have a Canadian passport*" and Q = "*You are a Canadian citizen*", then it is clear that this proposition is P → Q.

If both P and Q are false, then P → Q is simply true. There is nothing wrong in not having a Canadian passport while you are not a Canadian citizen. In fact, all those who are not Canadian citizens do not have a Canadian passport. Isn't this the way it should be?

When P is false and Q is true, then P → Q is also true. If you do not have a Canadian passport, you can still be a Canadian citizen. Not all Canadian citizens have Canadian passports; they only obtain one if they need to.

When both P and Q are true, then P → Q is also true. This does not require much explanation. If someone has a Canadian passport and is a Canadian citizen then the statement is true.

The last case is when P is true and Q is false. Here, P → Q is false. It can never be the case that someone has a Canadian passport, yet s/he is not a Canadian citizen. This is impossible and hence the proposition P → Q is false in this case.

Just like XOR, implication is redundant and can be expressed using the core logical operators: **not**, **and**, and **or**. The proposition P → Q is logically equivalent to (NOT P) OR Q. Verification using a truth table is left as an exercise.

Contrapositive: an implication P → Q is always equivalent to (NOT Q) → (NOT P); the latter is called the *contrapositive* of P → Q. This equivalence can be verified using a truth table and is left as an exercise.

Precedence of logical operators: If a complex proposition does not have brackets such as NOT Q AND B, we have to specify precedence rules so that the meaning of the proposition does not result in confusion. For instance does NOT Q AND B mean NOT (Q AND B) or does it mean (NOT Q) AND B. When there are no brackets, the operators are applied in the following order:

1. NOT
2. AND, OR, or XOR
3. Implication

That is, NOT Q AND B means (NOT Q) AND B because NOT has a higher precedence over AND and is applied first. When operators have equal precedence, they should be applied left-to-right as they appear in the proposition. For example, A AND B OR C is the same as (A AND B) OR C. The AND is applied first because it occurs first in the proposition.

To avoid such confusions, it is always a good idea to use brackets as we have been doing throughout our examples.

Tautologies and Contradictions: A *tautology* is a proposition that is always true. A *contradiction* is a proposition that is always false. A proposition that is neither a tautology nor a contradiction is called a *contingency*.

The simplest tautology is P OR (NOT P). This is going to be always true regardless of what the value of P is, as evident from the following truth table:

P	NOT P	P OR (NOT P)
F	T	T
T	F	T

The simplest contradiction is P AND (NOT P). This is going to be always false regardless of what the value of P is, as evident from the following truth table:

P	NOT P	P AND (NOT P)
F	T	F
T	F	F

Logical Inference: Inference is the process of building arguments and making valid conclusions starting from some hypotheses. We use inference all the time in our daily logic. If some knows that you have to pass a required course to graduate; and we also know that Mr. X did not pass a required course; then we can conclude that Mr. X is not graduating.

We do not intend to have a comprehensive coverage of inference rules here. Instead, we limit discussion to 3 famous examples of inference rules.

Modus Ponens:

A → B
A

B

Before explaining Modus Ponens, let's explain the notation being used. An argument consists of some propositions, written each on a separate line and a conclusion, written under the "-------". The Modus Ponens rule is read as follows: If we know that A implies B and we also know that A is true, then B must be true.

EXAMPLE 6 – Modus Ponens

Assume that P = "*Alice has a Canadian passport*" and Q = "*Alice is a Canadian citizen*".
P → Q : If Alice has a Canadian passport, then Alice is a Canadian citizen
P : Alice has a Canadian passport

Q : We can conclude that Alice must be a Canadian citizen

Note that in a rule of inference, such as Modus Ponens, the conclusion is true as long as the premises are true. In fact, Modus Ponens is based on the following tautology: [(A → B) AND A] → B. The verification is left as an exercise.

Modus Tollens:

A → B
NOT B

NOT A

The Modus Tollens rule is read as follows: If we know that A implies B and we also know that B is false (alternatively, NOT B is true), then A must be false. Modus Tollens is based on the following tautology: [(A → B) AND NOT B] → NOT A. The verification is left as an exercise.

EXAMPLE 7 – Modus Tollens

Assume that P = "*Alice has a Canadian passport*" and Q = "*Alice is a Canadian citizen*".
P → Q : If Alice has a Canadian passport, then Alice is a Canadian citizen
NOT Q : Alice is not a Canadian citizen

NOT P : We can conclude that Alice does not have a Canadian passport

Hypothetical Syllogism:

A → B
B → C

A → C

The Hypothetical Syllogism rule is read as follows: If we know that A implies B and we also know that B implies C, then A implies C. Hypothetical Syllogism is based on the following tautology: [(A → B) AND (B → C)] → (A → C). The verification is also left as an exercise.

EXAMPLE 8 – Hypothetical Syllogism

Assume that P = "*You have more than two drinks*", Q = "*You're unable to drive*", and R = "*You need a ride*".
P → Q : If you have more than two drinks, then you're unable to drive
Q → R : If you're unable to drive, then you need a ride

P → R : We can conclude that if you have more than two drinks, then you need a ride

EXAMPLE 9 – Logical inference

Given the hypotheses:
1. If you send me your part of the project, I will finish the project
2. If you do not send me your part of the project, I will go out drinking
3. If I go out drinking, I will sleep in
4. If I sleep in, I will miss my Computer Science class
5. You did not send me your part of the project

We can conclude that: "I will miss my Computer Science class".

Let S = "you send me your part of the project", F=" I will finish the project", D="I will go out drinking", I = "I will sleep in" and M = "I will miss my Computer Science class". Here is an argument:

Step	Explanation
1. S → F	Hypothesis(1)
2. (NOT F) → (NOT S)	Contrapositive of step 1
3. (NOT S) → D	Hypothesis(2)
4. (NOT F) → D	Hypothetical Syllogism from steps 2 and 3
5. D → I	Hypothesis(3)
6. (NOT F) → I	Hypothetical Syllogism from steps 4 and 5
7. I → M	Hypothesis(4)
8. (NOT F) → M	Hypothetical Syllogism from steps 6 and 7
9. NOT F	Hypothesis(5)
10. M	Modus Ponens 8 and 9

Predicate logic: We have argued earlier that statements involving variables are not propositions, if the value of the variable is unknown. For example, x > 0 is not a proposition. A statement of this form is called a *predicate*. A predicate has two parts: the variables involved and the statement about the variables.

EXAMPLE 10 – Predicates

- Let Positive(x) denote the statement x > 0. Positive(1) is true, but Positive(-6) is false.
- Let Greater(x,y) denote the statement x > y. Greater(1,2) is false, but Greater(2,1) is true.

Predicates can be made propositions by fixing the value of the variable, such as Positive(1). However, *quantification* is a very powerful way by which predicates are converted to propositions without fixing the variable values. There are two types of quantifiers: *universal* and *existential*. Quantification requires a *universe of discourse*, the collection of values from which a variable can be fixed. With the predicate Positive(x), the universe of discourse is assumed to be the collection of all numbers. If for instance, Fido is a dog, the statement Positive(Fido) does not make any sense. Fido is not in our universe of discourse.

Universal quantification: Let P(x) be a predicate, then ∀x P(x) is a proposition. ∀x P(x) is true when P(x) is true for all the values of x in the universe of discourse.

EXAMPLE 11 – Universal quantification

- If the universe of discourse is all numbers, then ∀x Positive(x) is false because not all numbers are greater than 0.
- Let RW(x) = "x can read and write" and U(x) = "x is a university student". If the universe of discourse is all human beings, then the following proposition is always true: ∀x (U(x) → RW(x)). In plain English, all humans who are university students can read and write. Note however, that the proposition ∀x (U(x) AND RW(x)) is always false, since it asserts that all humans are university students and that they can read and write.

Existential quantification: Let P(x) be a predicate, ∃x P(x) is a proposition. ∃x P(x) is true when P(x) is true for at least one value of x in the universe of discourse.

EXAMPLE 12 – Existential quantification

- If the universe of discourse is all numbers, then ∃x Positive(x) is true because there is at least one value in the universe of discourse, which is positive, say 1.
- Let RW(x) = "x can read and write" and U(x) = "x is a university student". If the universe of discourse is all human beings, then the following proposition is always false: ∃x (U(x) AND NOT RW(x)). In plain English, It is not possible to find a human who is a university student and cannot read and write (at least you hope so).

EXAMPLE 13 – Quantification

Universe of discourse: all earth creatures. Let M(x)= "x is a monkey" and F(x)= "x lives in a forest".

1. The following statement states that some monkeys live in forests: $\exists x$ M(x) AND F(x), and this proposition is true.
2. How would we express in logic, the statement all monkeys live in forests? One obvious attempt is: $\forall x$ M(x) AND F(x), but this is wrong. This statement says that all earth creatures are monkeys and live in forests! The proper way is: $\forall x$ M(x) $\rightarrow$ F(x), meaning that from all creatures if x is a monkey, then x lives in a forest. Since not all monkeys live in forests, the proposition $\forall x$ M(x) $\rightarrow$ F(x) is false.

Quantifier equivalence:

- $\forall x$ P(x) is equivalent to NOT $\exists x$ NOT P(x)
  Saying *all monkeys are black* is the same as saying *there is no one monkey which is not black*
- $\exists x$ P(x) is equivalent to NOT $\forall x$ NOT P(x)
  The statement *there is at least one student who likes the course* is logically the same as *it is not the case that all students do not like the course*

II – Naïve Set Theory

A *set* is an unordered collection of unique objects. The objects of a set are called *elements* or *members* of the set. Order of objects in a set is irrelevant. Also, the objects must be unique, that is, in a set an element cannot be repeated.

EXAMPLE 14 – Set notation

Small sets can be represented by listing its members

- A = {paper, scissors, rock}
  A is the set of all possible choices in the game
- B = {pawn, rook, knight, bishop, queen, king}
  B is the set of all chess pieces
- C = {1, 2, 3, 4, 5}

C is the set all positive integers less than or equal to 5

It is not always possible to list all members of a set. Some sets are very large (and some are infinite). Large sets can be represented using the set builder notation

- A = {x | x is a current student at UofC}
  A is the set of all current students at UofC
- B = {x | x is an even number}
  B is the set of all even numbers; this is an infinite set

EXAMPLE 15 – Duplicate elements

The set {paper, scissors, rock} is the same as the set {paper, scissors, rock, rock}. We should not repeat elements in a set.

EXAMPLE 16 – Order of elements

The order of members in a set is immaterial: {paper, scissors, rock}, {scissors, rock, paper}, and {rock, scissors, paper} all represent the same set.

Ordered tuples: We use curly brackets (braces) to represent sets, for instance {rock, paper} which is the same set as {paper, rock}. If the order of members is important we use *ordered tuples*. Ordered tuples are listed between parentheses. An example order pair (tuple of two members) is : (rock, paper). Note that (rock, paper) ≠ (paper, rock). Here the order of elements is important. These pairs could be representing the move choices in the paper-scissors-rock game of two players. (rock, paper) would mean that player 1's choice is rock, and player 2's choice is paper, while (paper,rock) would mean the opposite.

A tuple of size n has the general form: $(v_1, v_2, v_3, \ldots v_n)$. In tuples, repetition of elements is allowed. The tuple (90, 85, 90, 80, 95) could be the grades corresponding to the project grades of a team of 5 students. Students 1 and 3 have the same grade.

Set inclusion: A set A is said to be a *subset* of set B, written A $\subseteq$ B, if all the elements of A are also elements of B.

EXAMPLE 17 – Set inclusion

- {1,2} $\subseteq$ {1,2,7}
- {a} $\subseteq$ {x | x is a letter in the English alphabet}
- {H,I,T} $\subseteq$ {H,I,T}

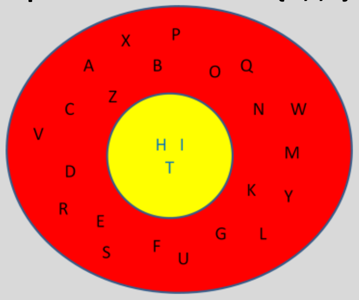
Note that element membership in a set is denoted differently: H $\in$ {H,I,T} signifies that H is an element of the set {H,I,T}, but the set notation {H} $\subseteq$ {H,I,T} says that the set containing the single element H is a subset of {H,I,T}.

Venn Diagrams are a pictorial way of representing sets. A set is drawn as an ellipse and the member elements are indicated inside the ellipse.

EXAMPLE 18 – Venn diagrams (inclusion)

The following Venn diagram shows the set of English alphabets (in red) and the letter of the word HIT for another set which is simply a subset of the English alphabets.

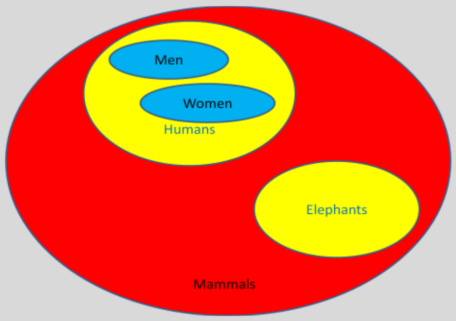
Venn diagram showing the set of English alphabets and the set {H,I,T}



EXAMPLE 19 – Venn diagrams (inclusion)

This partial example of the set of mammals group humans as a subset of mammals. Elephants are also a subset of mammals. Humans can be further portioned into men and women.

Venn diagram showing the relationship between mammals, humans, and elephants



Set intersection: The set A ∩ B = set of all elements that are common between sets A and B. More formally, A ∩ B = { x | x ∈ A AND x ∈ B}.

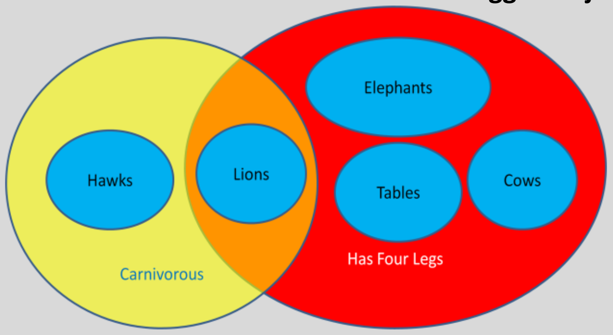
EXAMPLE 20 – Set intersection

Let A = {1, 6, 8}, B = {1, 3, 5, 7}, C = {3, 5, 7}
A ∩ B = {1}
B ∩ C = {3, 5, 7} = C
A ∩ C = {}, the empty set, also denoted ϕ

EXAMPLE 21 – Venn diagrams (intersection)

Lions belong to the intersection of carnivorous creatures and objects with four legs, Hawks and cows do not

Venn diagram showing partial relationship between carnivorous creatures and four-legged objects



Set union: The set $A \cup B$ = set of all elements that are in A plus these that are in B. More formally, $A \cup B = \{ x \mid x \in A \text{ OR } x \in B\}$.

EXAMPLE 22 – Set union

Let $A = \{1, 6, 8\}$, $B = \{1, 3, 5, 7\}$, $C = \{3, 5, 7\}$
$A \cup B = \{1, 3, 5, 6, 7, 8\}$
$B \cup C = \{1, 3, 5, 7\} = B$
$A \cup C = \{1, 3, 5, 6, 7, 8\}$

Set difference: The set $A – B$ = set of all elements that are in A minus these that are in B. More formally, $A - B = \{ x \mid x \in A \text{ AND } x \notin B\}$ (note that NOT $(x \in B)$ is denoted $x \notin B$).

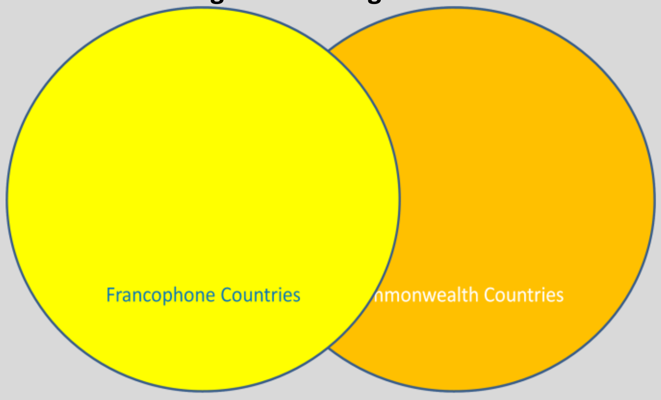
EXAMPLE 23 – Set difference

Let $A = \{1, 6, 8\}$, $B = \{1, 3, 5, 7\}$, $C = \{3, 5, 7\}$
$A = \{1, 6, 8\}$, $B = \{1, 3, 5, 7\}$, $C = \{3, 5, 7\}$
$A – B = \{6, 8\}$
$B – C = \{1\}$
$C – B = \{\} = \phi$
$C – A = C$

EXAMPLE 24 – Venn diagrams (difference)

Given that some countries are Francophone and also part of the Common Wealth, such as Canada, The set of Common Wealth countries that are not Francophone is simply the former set minus the latter (indicated in Orange)

Venn diagram showing set difference



Set multiplication: The set $A \times B = \{ (a,b) \mid a \in A \text{ AND } b \in B\}$. In general $A_1 \times A_2 \times \ldots \times A_n = \{(a_1,a_2,\ldots, a_n) \mid a_1$ is in A_1 and a_2 is in $A_2 \ldots a_n$ is in $A_n\}$

EXAMPLE 25 – Set multiplication

1. Let A = {a} and B = {1, 2}

A × B = {(a,1), (a,2)}

2. Let A = {player1, player2} and B = {paper, scissors, rock}

A × B = {(player1, paper), (player1, scissors), (player1, rock),
(player2, paper), (player2, scissors), (player2, rock)}

Relations: A *relation* from A to B is a subset of A × B. In general, A relation on A_1 , A_2 , … , A_n is a subset of A_1 x A_2 x … x A_n .

EXAMPLE 26 – Relations

Let A = {x | x is a UofC student}
Let B = {y | y is a Ferrari car}
A × B = {(x,y)| x is in A and y is in B}
R = {(x,y) | x is a UofC student and x owns the Ferrari y}

Exercises

1. Use truth tables to show that the following propositions are equivalent
   a. A OR (B AND C) = (A OR B) AND (A ORC)
   b. A AND (B OR C) = (A AND B) OR (A AND C)
2. What are the truth values of the following propositions?
   a. If you get an A, then you pass the course
   b. If you pass the course, then you get an A
   c. If you're a citizen, then you vote in the elections
   d. If you do not pass the course, then you do not get an A
3. Prove the following argument
   1. If I see you, I will be happy
   2. If I am happy, I dance
   3. I did not dance
   4. Therefore, I did not see you tonight
4. Write logical expressions for the following propositions. The universe of discourse is all Human beings
   a. All men are mortal
      Dictionary, M(x): x is a man, O(x): is mortal
   b. Some women are immortal
   Dictionary, W(x): x is a woman, O(x): is mortal
   c. All monkeys live in forests or zoos
   Dictionary, M(x): x is a monkey, F(x): x lives in a forest, Z(x): x lives in a zoo

Chapter 2
Making Computers Think: Programming

Computers, just like other machines, transform an input to an output. A washing machine transforms dirty clothes, the input, to clean clothes, the output. Unlike other machines, computers are versatile; they can be programmed to solve almost any problem. Computers solve problems by following concrete instructions that outline step-by-step what needs to be done. The problems solved by computers are called *computational problems*, and the concrete instructions they follow to solve these problems are called *programs*. A computational problem is a specification of the relationship between the input and output. Often, this is called the *what*. That is computational problems specify what needs to be done.

While the problem specifies the *what*, computers need specific step-by-step instructions to know *how* to convert some input to some output. An *algorithm* is a specification of such *how*. Programs are a specific implementation of algorithms, using a specific programming language. The language we will study here is called Jython, it is based on the Python language, but is implemented in Java.

I –Algorithms and Programs

Algorithms can be specified in different ways: from pseudo-code to programs. Here, we focus on how to implement algorithms using programs.

Computers only understand 0s and 1s and the language they understand is called *machine language*. Programming in machine language is too difficult if not impossible. Instead, programmers write their code in a *high-level* (HL) programming language. HL languages are much easier to program with since they allow us to write a program using English instructions. Yet, they are formal enough to be converted to machine language, or *low-level* (LL) code. Different machines may have different LL languages, depending on the way the machine is built.

The conversion from HL to LL programs is called *translation*. Translators are programs that carry on the conversion. The input to a translator is a HL program and the output is an equivalent LL program. There are two types of translation mechanisms: *Compilation* and *interpretation*. Both do the same job, converting HL to LL code, but they do it differently. Compilers read the whole HL program and translate it entirely to a LL program before the LL program is executed. Interpreters read one HL instruction at a time, translate it to LL, and execute it. There are advantages and disadvantages for each method, which will not be discussed here. Some programming languages, such as Java, use a combination of compilation and interpretation.

Whether a program is compiled or interpreted, it is executed one instruction at a time. The execution of an instruction in the program does not commence until the immediately preceding instruction has been completed. Programming languages (HL or LL) have grammars that need to be followed. Just like in English or French, a sentence has a particular grammatical structure, programs follow grammatical rules. The difference is that program grammars are very strict and must be followed to the letter, for the program to be translated or executed. In what follows, we will study the basic parts of speech of a simple programming language called Jython. Jython is an interpreted language.

II – Programming in Jython

Variables and assignments: One central concept in programming is the notion of a *variable*. Computers need to memorize data to be able to work on them. That is computers store data in memory so that they can process them. Variables allow us to abstract memory by giving symbolic names to memory locations, the places where data is held. The values in these variables are set using a simple instruction called *assignment*.

The names you give to variables can consist of any letters and numbers, as long as the name starts with a letter, such as number1, num2, and name.

EXAMPLE 1 – Assignments

x = 10
x is a variable and the equal sign = is assigning 10 to x. This is read: x is assigned 10, or x gets 10. A variable like x, defines a place holder, a location where values can be stored. The effect of the statement **x = 10** is to store in the place holder x the value 10.

y = 12
z = x+y
An assignment works by first evaluating the left hand side of **=**, in order to know what value needs to be stored, and then assigning the resulting value in the variable on the left hand side. The left hand side must be always a variable, but the right hand side can be anything that generates a value, such as x+y. So, the value stored in variable z will be the value in x plus the value in y, which is 22

y = x
y += 5
Each time a variable is assigned a new value using =, it loses the old value. Effectively, the new value overwrites the old one. So, **y = x** results in storing 10 in y. The assignment **+=** is different. The statement **y += 5** instructs the computer to add 5 to y, instead of replacing the old value. The new value in y will be 15. This is the same as writing **y = y + 5**. Similarly, we can use -= to subtract a value from a variable, *= to multiply the variable by some value, and /= to divide it by a value.

Basic data types: variables have data types. The age of a person should be of type number, but the name should be of type string (text). There are four basic data types in Jython:

- *Integer* is for numbers that do not contain fractions (example values, -1, 9, 1119, and -19191919)
- *Float* is for numbers that may contain fractions (such as 1.22, -100.5, -0.0001, and 1.0)
- *String* is for text. The string values must be enclosed with single or double quotes. (examples: ‘Hi’, “Hello There”, “Welcome player 1 …”, ‘Welcome to CPSC 203’). For consistency, we will be always using single quotes to enclose string values.

- *Boolean* type has two values *true* or *false*. (true can be also represented by 1 and false by 0)

The type of the variable is determined by the value it stores.

Arithmetic operations: Arithmetic operations can be performed on number types (integer and float) and include:

- Addition: **y = x+1**
- Subtraction: **diff = num1 – num2**
- Multiplication: **result = interest * principal**
- Division: **x = y / 2**
- Modular Division: **x = y % 2**

Example 2 – Division

y = 7
x = y / 2
The value of x will be 3.5 and x is of type float.

y = 7
x = y % 2
The modular division %, simply returns the remainder of the division of y % 2. Since y is 7, the reminder of dividing 7 by 2 is 1 (quotient is 3 and reminder is 1). The value in x is an integer.

Conversion between data types: Jython allows us to convert a value in a given data type to another. For instance, the value '12' is a string (notice the quotes), we can convert it to an integer as follows: **x = int('12').** The value in x would be integer 12. As another example, the statement **x = int(7/2)** results in storing 3 in x (the integer part of 3.5). Conversion to float can be used similarly. For instance, **x = float('12')** results in storing the floating point number 12.0 in x.

Output: programs need to communicate with the user. One way in jython to output a message to the user is the **print** statement. The print statement prints a message on the screen on a separate line.

EXAMPLE 3 – print statement

print 'hello there'
Prints the string enclosed by single-quotes as is.

print 5*4
Prints 20

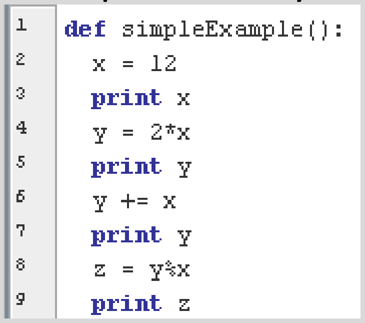
print '5*4'
Prints 5*4

num = 12
print 'The number is' , num
Prints The number is 12

Functions: Program statements can be grouped together into a *function*. A typical program may consist of several functions.

EXAMPLE 4 – Functions

An example function in Jython



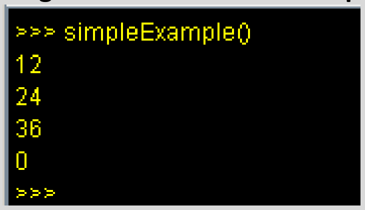
```
1  def simpleExample():
2    x = 12
3    print x
4    y = 2*x
5    print y
6    y += x
7    print y
8    z = y%x
9    print z
```

The line numbers in the above listings are not part of the program; we will always include line numbers with our example programs to allow us to reference certain statements. Line 1 is defining a function header. The header consists of the word **def**, the function name, **simpleExample**, and a colon, **:**. The word **def** is always used to define a function. Function names must always use brackets. The colon indicates the end of the header or the beginning of the function *body*. The body of the function starts at line 2. The statements on lines 2 to 9 are indented with a tab to indicate inclusion in the function body.

The words shown in bold blue face, **def** and **print**, are *reserved* words. These have special meaning to the programming language and we must not use them (as is) to define variables or function names, otherwise the interpreter will be confused.

This function can be executed by calling it by name, which triggers the execution of the statement at line 2, the first statement in the body. After line 2 is executed, the statement on line 3 is executed, resulting in printing 12 on the screen. Then the statement at line 4 is executed and hence forth until the last statement in the function is executed. When the last statement in a function finishes execution, we say the function *returns* or *exits*.

Calling the function and its output



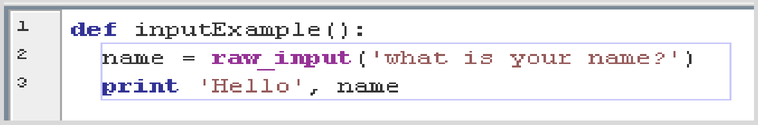
```
>>> simpleExample()
12
24
36
0
>>>
```

Keyboard input: Often programs need to collect some input from the user, such as typing something on the keyboard. In Jython this can be done using the built in function:

raw_input('type in some message here')

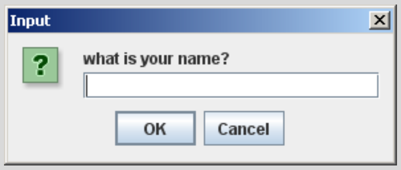
This is a library function, also called a *built-in* function. Such functions can be readily used by you. The raw_input function creates a window through which it can collect some input from you. This function *returns* the keyboard input, meaning that if it is used as the right hand side of an assignment statement, it will be replaced by the value entered by the user.

EXAMPLE 5 – Raw input

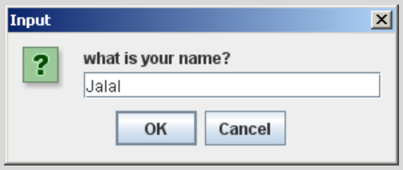


```
def inputExample():
    name = raw_input('what is your name?')
    print 'Hello', name
```

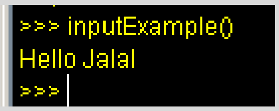
The function inputExample calls raw_input at line 2, which creates the following window:



When the user enters a value and clicks 'OK', the function raw_input stops execution and is replaced on line 2 by the value entered by the user. For instance, if I enter my name:



And I press 'OK', line 2 of the function becomes equivalent to the statement: **name = 'Jalal'**. That is, raw_input returns the value 'Jalal' as entered by the user. A sample output is as follows:



```
>>> inputExample()
Hello Jalal
>>>
```

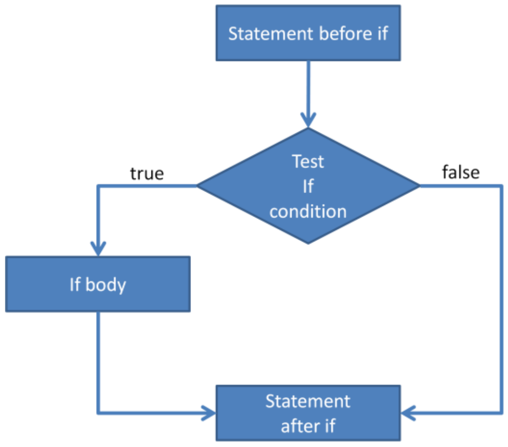
Conditionals: The programs we have considered so far are *straight-line* code. Often, programs need to make decisions. The program of an ATM machine checks to see if you have enough money in the account before dispensing any cash. If you have enough money to make a withdrawal, the ATM still needs to verify that you are not exceeding your daily limit. That is, the output of the ATM depends on making some decisions and it could be some cash or a sorry message. This kind of execution is called branching and the statements that allow a program to behave in such a way are called conditional statements.

If conditionals: The simplest form of conditional statements is an *if* statement. The general structure of an if statement is as follows:

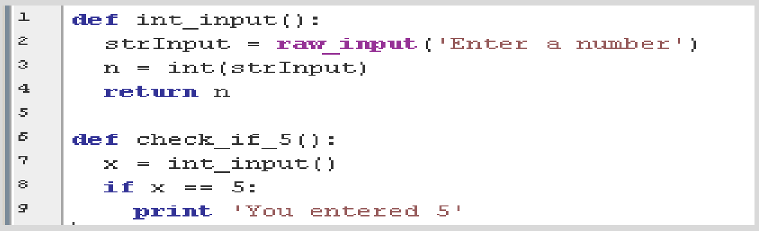
if *Boolean condition* :
Body of if

The Boolean condition is tested. If it is false, the body of the if will be skipped all together. If it is true, the body of the if is performed. In either case, execution continues with the statement that immediately follows the if.

Flow-charts: A flow-chart is a visual representation of the program flow. Flow charts can be very useful in specifying algorithms as well. The flow charts that we will use here are very simple. Boxes indicate statements that do not involve decisions, such as an assignment or a print statement. We use a diamond shape to indicate a decision, with two possible outcomes, either true or false. The flow of an if statement is depicted in the following flow-chart.



EXAMPLE 6 – If statement



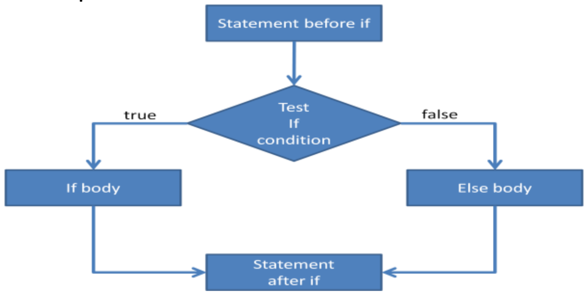
```
1  def int_input():
2     strInput = raw_input('Enter a number')
3     n = int(strInput)
4     return n
5
6  def check_if_5():
7     x = int_input()
8     if x == 5:
9       print 'You entered 5'
```

This program contains two functions **int_input()** and **check_if_5()**. The int_input function collects a number from the user as a string (since raw_input returns a string), converts it to an integer (line 3), and returns the integer value at line 4. This is how functions return values, so that when it is called by check_if_5 at line 7, the value stored in x will be the integer that the user entered. The if statement itself is defined on lines 8 and 9. Line 8 is the header of the if, which contains the *if* keyword, followed by a Boolean condition, followed by a colon, **:**. The == symbol must be differentiated from the assignment statement, a single =. **x == 5** is a Boolean condition that evaluates to true (if x is equal to 5) or false (otherwise), depending on the value of x. An if body can contain many statements and they have to be indented from the if header, just like the case with function bodies and headers.

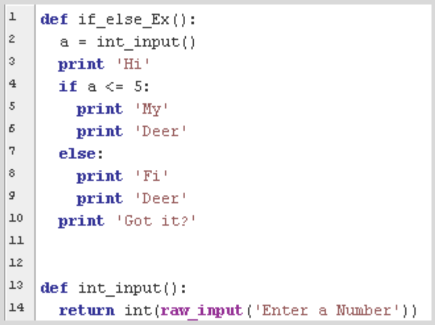
If-Else Conditionals: An *if-else* statement has an if and else parts. The general structure of an if-else statement is as follows:

> ***if** Boolean condition :*
>   *Body of if*
> ***else:***
>   *Body of else*

The Boolean condition is tested. If it is false, the body of the if will be skipped all together, but the body of the else will be performed. If it is true, the body of the if is performed and the body of the else is skipped. In either case, execution continues with the statement that immediately follows the if-else. The next flow chart depicts an if-else operation.



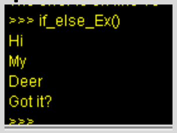
EXAMPLE 7 – If-else statement



```
1  def if_else_Ex():
2    a = int_input()
3    print 'Hi'
4    if a <= 5:
5      print 'My'
6      print 'Deer'
7    else:
8      print 'Fi'
9      print 'Deer'
10   print 'Got it?'
11
12
13 def int_input():
14   return int(raw_input('Enter a Number'))
```

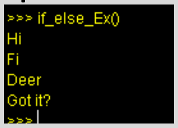
The if-else statement in the above listing spans lines 4 to 9. The statement immediately preceding the if-else is at line 3, **print 'Hi'**, which will be always executed before the if-else. At line 4 **a** is checked to see if it is less than or equal to 5. If this is the case, the body of the if (lines 5 and 6) is executed and the else body (lines 8 and 9) is skipped. Otherwise (**a** is greater than 5), then the if (lines 5 and 6) is skipped and the else body (lines 8 and 9) is executed. In either case the print at line 10 will be executed after the if-else.

Output when a <= 5



```
>>> if_else_Ex()
Hi
My
Deer
Got it?
>>>
```

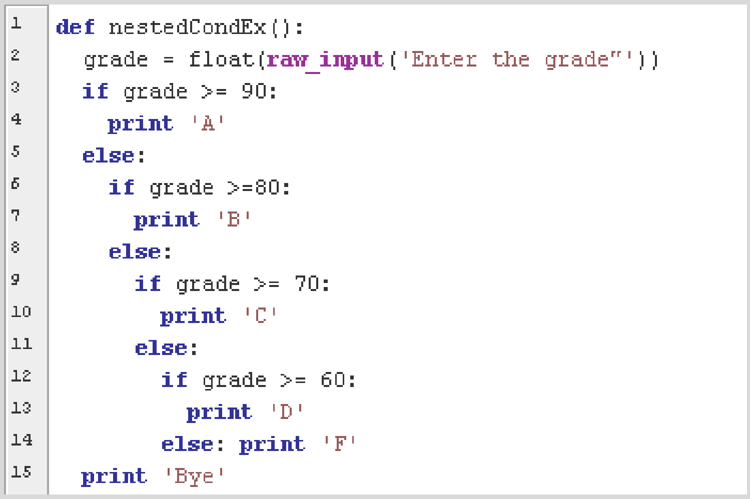
Output when a > 5



```
>>> if_else_Ex()
Hi
Fi
Deer
Got it?
>>>
```

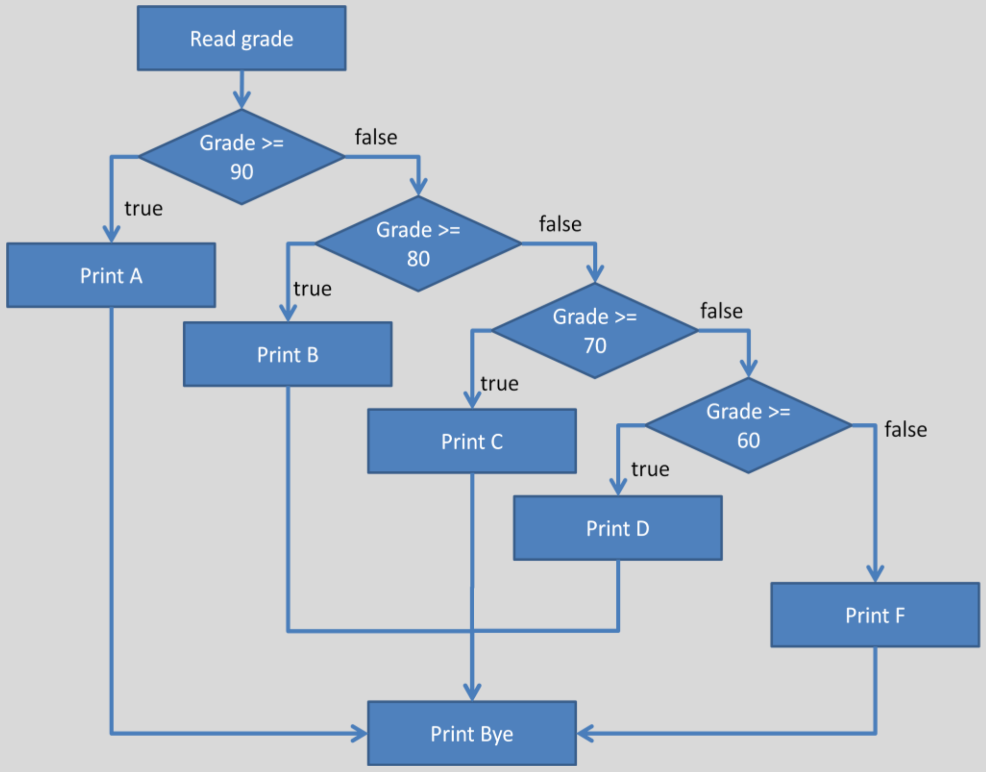
Nested conditionals: An else statement may also need to check a condition to decide how to proceed. This can be done by including another if-else statement inside the else, called *nested* if-else.

EXAMPLE 8 – Nested if-else



```
1  def nestedCondEx():
2    grade = float(raw_input('Enter the grade"'))
3    if grade >= 90:
4      print 'A'
5    else:
6      if grade >=80:
7        print 'B'
8      else:
9        if grade >= 70:
10         print 'C'
11       else:
12         if grade >= 60:
13           print 'D'
14         else: print 'F'
15   print 'Bye'
```

In this listing the grade is read as a floating point number on line 2. The rest of the function is a single if-else statement. The else (at line 5), though, contains another if-else (lines 6 to 14). Here, also, the else contains another if-else (lines 9 to 14), and hence forth. The following flowchart explains the operation of this program

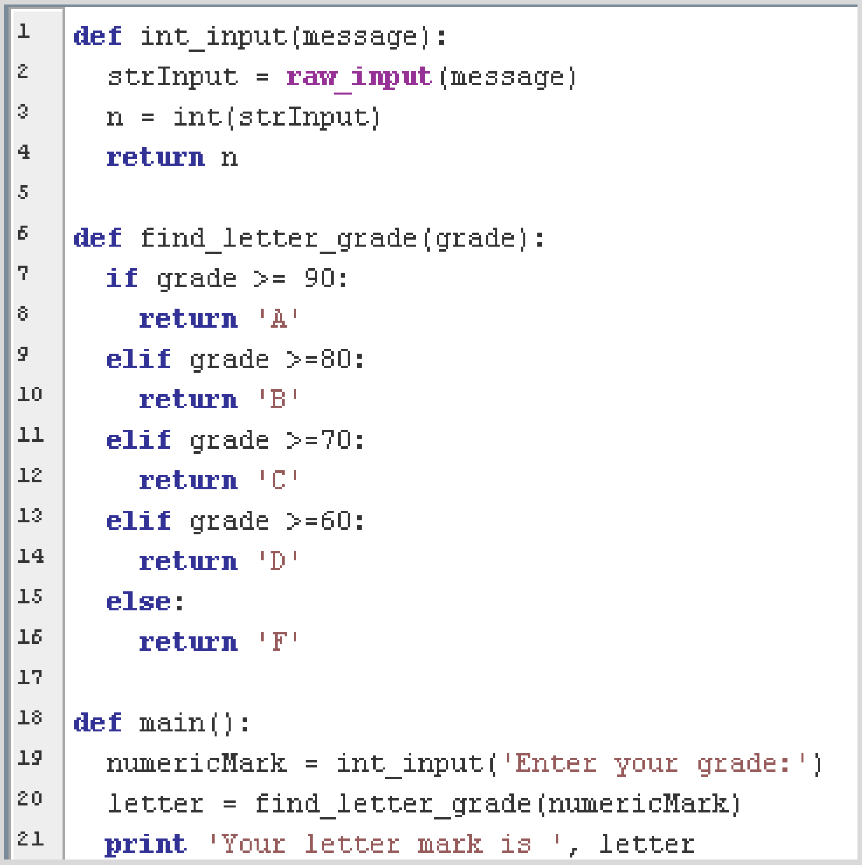


After reading the grade, if it is ≥ 90, the program prints **A**, **Bye**, and quits. Otherwise, the false branch of **Grade >= 90** is followed and takes us to another condition **Grade >= 80**. If this condition is satisfied, the program prints **B**, **Bye**, and quits. Otherwise, the false branch of **Grade >= 80** is followed and takes us to another condition **Grade >= 70**. If this condition is satisfied, the program prints **C**, **Bye**, and quits. Otherwise, the false branch of **Grade >= 70** is followed and takes us to another condition **Grade >= 60**. If this condition is satisfied, the program prints **D**, **Bye**, and quits. Otherwise, the false branch of **Grade >= 60** is followed and the program prints **F**, **Bye**, and quits.

Note that When the false branch of **Grade >= 90** is followed, it must be the case that **Grade < 90**. Hence, it is sufficient to check if **Grade >= 80** to print a B, since (according to this program, of course) a B is between 80 (inclusive) and 90 (exclusive).

Jython offers and abbreviation to an else statement that contains an if, **elif**. The next example shows a similar program.

EXAMPLE 9 – Nested if-else



```
1  def int_input(message):
2    strInput = raw_input(message)
3    n = int(strInput)
4    return n
5
6  def find_letter_grade(grade):
7    if grade >= 90:
8      return 'A'
9    elif grade >=80:
10     return 'B'
11   elif grade >=70:
12     return 'C'
13   elif grade >=60:
14     return 'D'
15   else:
16     return 'F'
17
18 def main():
19   numericMark = int_input('Enter your grade:')
20   letter = find_letter_grade(numericMark)
21   print 'Your letter mark is ', letter
```

The function **int_input** has an *argument* or *parameter*, **message**. This is a variable that can be used by the function only. It is different from other variables in the sense that it is assigned a value when the function is called, **int_input('Enter your grade:')** (line 19). The value **'Enter your grade'** is assigned to the variable **message** used in function **int_input**.

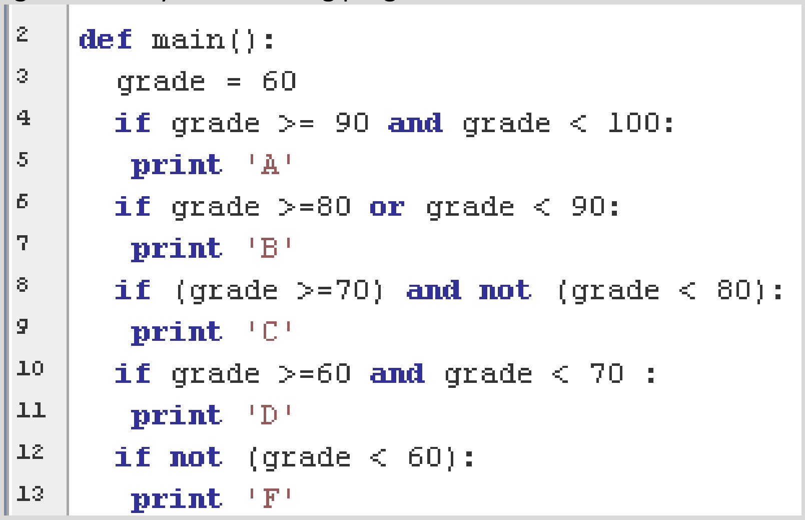
This is also the case with function **find_letter_grade**; the argument here is **grade**, and is assigned a value at line 20.

Note that this program accepts grades that are more than a 100. How would you disallow this?

Boolean conditions: More complex Boolean conditions can be constructed, using the logic operators **and**, **or**, and **not**.

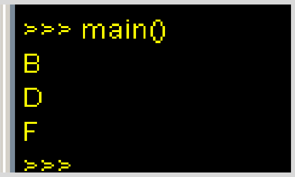
EXAMPLE 10 – Boolean Conditions

What is the output generated by the following program?



```
def main():
  grade = 60
  if grade >= 90 and grade < 100:
   print 'A'
  if grade >=80 or grade < 90:
   print 'B'
  if (grade >=70) and not (grade < 80):
   print 'C'
  if grade >=60 and grade < 70 :
   print 'D'
  if not (grade < 60):
   print 'F'
```

The answer is:



```
>>> main()
B
D
F
>>>
```

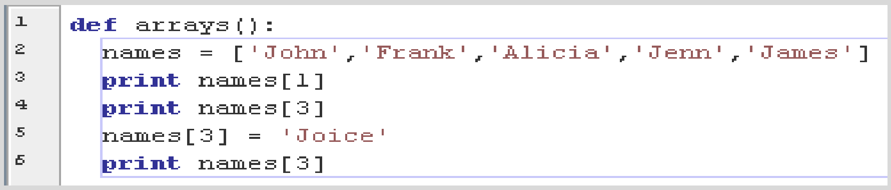
Arrays and lists: An *array* is a list of numbered (indexed) elements. Lists in Jython have many built-in functions that could be useful. One of these functions is len(), which returns the length of the list. The length of a list is the number of elements it contains.

EXAMPLE 11 – Arrays

Names = [‘John’, ’Frank’, ‘Alicia’, ‘Jenn’, ‘James’]
This Jython statement is assigning to **Names** a value which is a list of names. These names are numbered in order, left to right. Numbering always starts at 0. So, John has the index 0, Frank has the index 1, …, and James has the index 4.

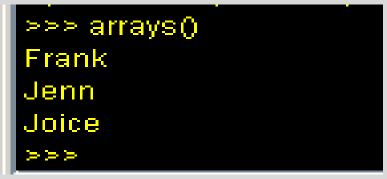
print len(Names)
This Jython statement prints 5, since **Names** contains 5 values.

An array should be viewed as a collection of variables bundled together. The individual variables can be accessed and used just like any other variable. To reference them, we need to specify their index. **Names[1]** is a variable that refers to cell 1 in array **Names**, which contains Frank. Similarly, **Names[3]** contains Jenn. The following listing shows how these can be used as regular variables



```
def arrays():
  names = ['John','Frank','Alicia','Jenn','James']
  print names[1]
  print names[3]
  names[3] = 'Joice'
  print names[3]
```

The output to this program is:



```
>>> arrays()
Frank
Jenn
Joice
>>>
```

Loops: loops are programming structures that allow us to repeat some statements as many times as required. We will discuss two loop types: *for* and *while* loops. For loops are suitable when you need to repeat something for a known number of times. While loops are used when we need to repeat something until a certain condition is met.

For loop: For loops in Jython can be used with lists or with the *range* function. The next example entertains the first possibility

EXAMPLE 12 – For loops with lists

for i in [0,1,2,3,4,5,6,7,8,9]:
  print i
A for loop is defined by the **for** keyword, followed by a variable, called the *index* variable; after the index variable the **in** construct occurs, followed by a list of elements. Logically, the for header says that *for each value in the list do the body of the for loop*. Since the list **[0,1,2,3,4,5,6,7,8,9]** has 10 values, we expect the body of the for loop, which is **print i** in our case, to be repeated 10 times. We say the loop *iterates* 10 times and each time it is repeated is called an *iteration*. The index variable **i** is initially assigned the value 0, the first value in the list. So, in the first iteration of the for loop, the value of **i** is 0.

That is, the first thing that is done is printing 0. Once the body finishes executing the first iteration, we check to see if the value assigned to **i** is the last value in the list. If this is the case, the loop ends and execution moves to the next statement in the program, the one that immediately follows the loop. In our case, the value of **i** is 0 and 0 is not the last element in the list. In this case, **i** is assigned the next value in the list, the value **1**, and henceforth. Therefore, the above loop prints the values 0 to 9.

```
for i in [0,1,2,3,4,5,6,7,8,9]:
  print 2*i
```

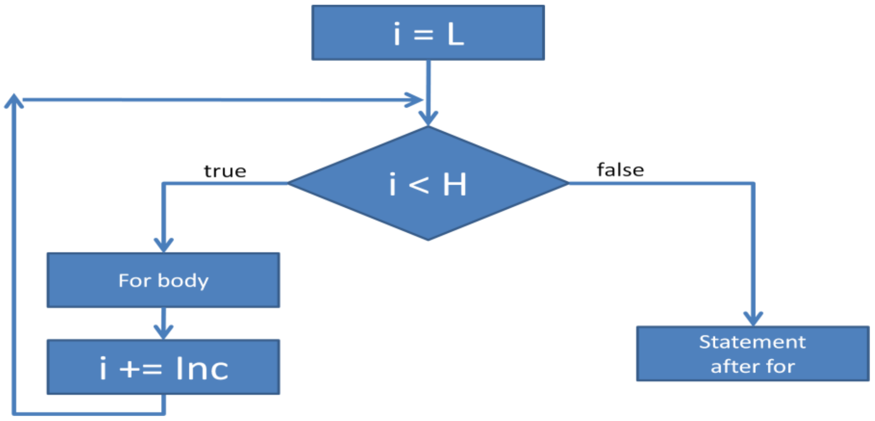
In this example, the loop prints 0, 2, 4, 6, 8, 10, 12, 14, 16, and 18, or the even numbers from 0 to 18.

```
for i in [9,2,7]:
  print i+1
```

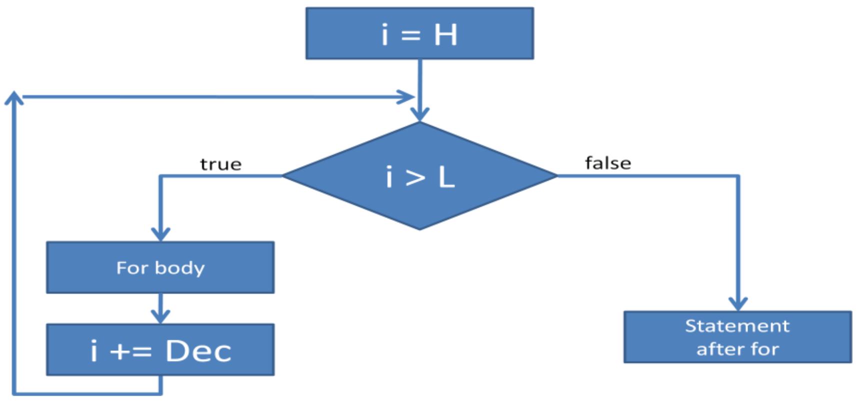
In this example, the loop prints 10, 3, and 8.

The **range(L,H)** function requires two arguments L and H and works as follows. If L < H , it generates values from L to H -1 in increments of 1. For instance, **range(0,9)** generates values from 0,1,2,3,4,5,6,7, and 8. The function **range(2,0)** generates no values, since it is not the case that 2 < 0 .

The general form of the range function allows a third argument: **range(L,H,Inc)**. The third argument **Inc** is the increment. When the increment is omitted, such as in **range(0,9)**, then it is understood that the increments are of 1. The function **range(0,9,2)** explicitly specifies the increments to be of 2; it generates values from 0 to 8 in increments of 2, or the list 0, 2, 4, 6, 8. In general if Inc is positive, the function **range(L,H,Inc)** generates values from L to H-1 in increments of Inc, as long as L < H. The following flowchart explains the operations of the loop **for i in range(L,H,Inc)**, when Inc is positive:



The function can also count backwards, **range(H,L,Dec)**. To count backwards, H > L and Dec < 0. For instance, **range(4,1,-1)** generates 4, 3, and 2. A flow chart for the operation of the loop: **for i in range(H,L,Dec)** is as follows:



EXAMPLE 13 – For loops with range

```
for i in range(0,5):
  print i
```
This loop prints 0, 1, 2, 3, and 4.

```
for i in range(0,5,3):
  print i
```
This loop prints 0, and 3. The next value would be 6, but 6 > 5.

```
for i in range(0,5, -1):
  print i
```
This loop prints nothing.

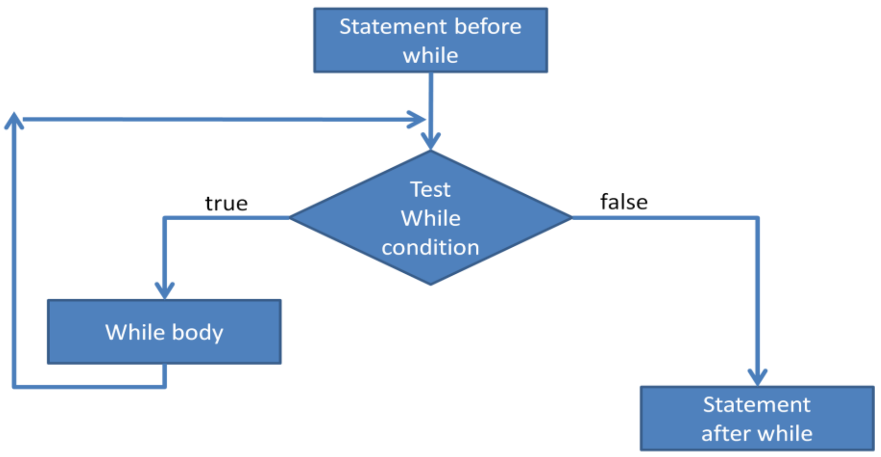
```
for i in range(5,0,-1):
  print i
```
This loop prints 5, 4, 3, 2, and 1.

```
for i in range(15,0,-5):
  print i
```
This loop prints 15, 10, and 5.

While loop: The while loop works in a similar way to an if statement. It has the following general structure:

while *Boolean-condition*:
Body-of-while

The *Boolean-condition* is checked first. If it is true, the *Body-of-while* is executed. Before the beginning of every iteration, the *Boolean-condition* is checked and the loop keeps iterating until the condition becomes false. The following flowchart explains the while loop operation:



EXAMPLE 14 – While loop

```
i = 0
while i < 10:
  print i
  i+=2
```

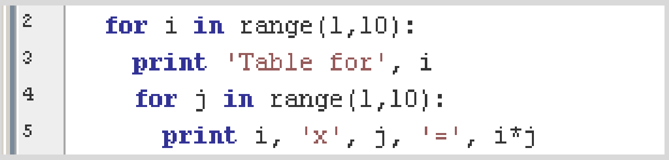
Since i is initially 0, the Boolean condition **i < 10** is true. So, the body of the while is executed, printing 0 and changing the value of i to 2.
Before the second iteration, the condition is checked again: i now is 2 and 2 < 10. The loop performs a second iteration printing 2 and changing i to 4.
Before the third iteration, the condition is checked again: i now is 4 and 4 < 10. The loop performs a third iteration printing 4 and changing i to 6.
Before the fourth iteration, the condition is checked again: i now is 6 and 6 < 10. The loop performs a fourth iteration printing 6 and changing i to 8.
Before the fifth iteration, the condition is checked again i now is 8 and 8 < 10. The loop performs a fifth iteration printing 28 and changing i to 10.
Before the sixth iteration, the condition is checked again i now is 10 and it is not the case that 10 < 10. The loop ends (there is no sixth iteration).

```
i = 5
while i > 0:
  print i
  i-=2
```

This loops prints 5, 4, 3, 2, and 1.

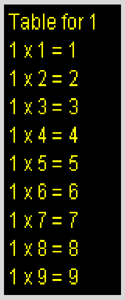
Nested loops: Loops can contain other loops; such loop structure is called *nested loops*.

EXAMPLE 15 – Nested loops



```
2  for i in range(1,10):
3    print 'Table for', i
4    for j in range(1,10):
5      print i, 'x', j, '=', i*j
```

This example listing has the loop at line 4 contained in the loop at line 2. The outer loop (line 2) performs 9 iterations (with i ranging from 1 to 9). In each of these iterations, the inner loop (line 4) performs 9 iterations. That is the print statement at line 5 is performed 9x9 = 81 times.
In fact, this listing prints the multiplication table. When i is 1, it prints the multiplication table for 1. The print statement at line 5 generates the output:



```
Table for 1
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
```

In the next iteration of the outer loop, i will be 2 and the inner loop prints the multiplication table for 2 and henceforth.

III – Searching and Sorting

The search problem: recall that a computational problem is specified as a relationship between input and output. The search problem is specified as follows:

Input: a list of *n* elements [a_0, a_1, a_2, … , a_{n-1}] and one additional element *x*
Output:
YES, if *x* is in the list [a_0, a_1, a_2, … , a_{n-1}]
NO, otherwise

EXAMPLE 16 – Search problem

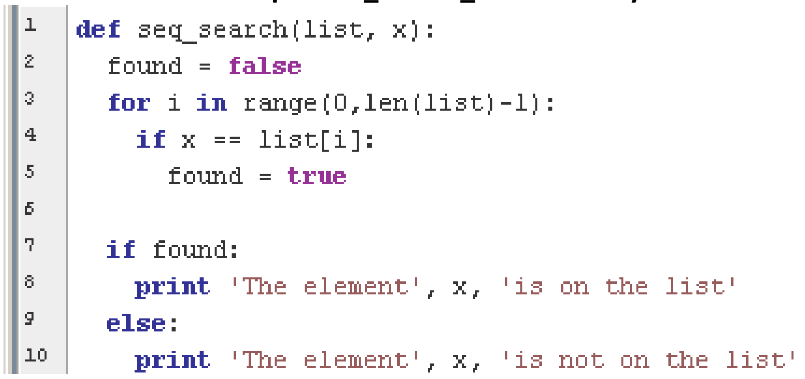
Input: [12, 5, 6, 100, 3, 1], 12
Output: YES

Input: [12, 5, 6, 100, 3, 1], 2
Output: No

Sequential Search Algorithm: The sequential search algorithm solves the search problem by examining each element in the list in order until it either finds the element or it goes through the whole list without finding it. Our first version of the algorithm is as follows:

```
Sequential_Search_Version1
    1. Found = NO
    2. i = 0
    3. Compare x with a_i
    4. If x == a_i Found = YES
    5. i += 1
    6. Repeat steps 3 to 5 until all elements are examined
    7. Print Found
```

Sequential_Search_Version1 in Jython



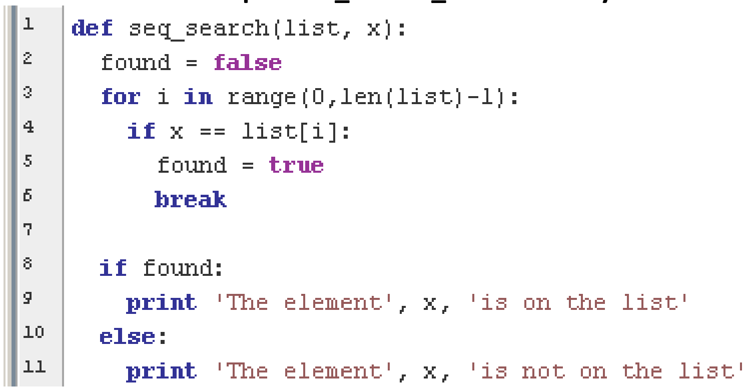
```
def seq_search(list, x):
  found = false
  for i in range(0,len(list)-1):
    if x == list[i]:
      found = true

  if found:
    print 'The element', x, 'is on the list'
  else:
    print 'The element', x, 'is not on the list'
```

This version does not stop the loop when the element is found. It keeps on checking the rest of the list, even though the result has already been determined. This is an unnecessary work. Version2 fixes this problem.

Sequential_Search_Version2

```
1.  Found = NO
2.  i = 0
3.  Compare x with $a_i$
4.  If x == $a_i$ Found = YES, Skip to step 7 (**STOP the loop**)
5.  i += 1
6.  Repeat steps 3 to 5 until all elements are examined
7.  Print Found
```

Sequential_Search_Version2 in Jython

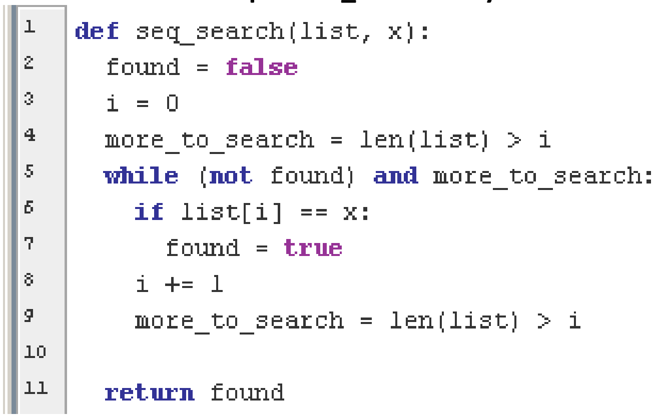


```
1   def seq_search(list, x):
2     found = false
3     for i in range(0,len(list)-1):
4       if x == list[i]:
5         found = true
6         break
7
8     if found:
9       print 'The element', x, 'is on the list'
10    else:
11      print 'The element', x, 'is not on the list'
```

The above listing introduces the **break** statement at line 6. The break statement simply breaks the nearest loop in which it is included. That is, when x is found in the list, the for loop is stopped.

The last (and cleanest) version of sequential search will make use of a while loop.

Sequential_Search in Jython



```
1   def seq_search(list, x):
2     found = false
3     i = 0
4     more_to_search = len(list) > i
5     while (not found) and more_to_search:
6       if list[i] == x:
7         found = true
8       i += 1
9       more_to_search = len(list) > i
10
11    return found
```

This version makes use of the Boolean variable **more_to_search**, which is intilaized at line 4 to **len(list) > i**. Since i is initially 0, **more_to_search** will be false only if **list** is empty, in which case **len(list)** is 0. Initially, we have elements to search if the list is not empty. The while loop on line 5 says that the search will be carried on as long as x is not found and we have elements to search. Since found is initially false, **not found** will be true. Once the element is found, **found** is set to true at line 7 and this will cause the search to stop since at the beginning of the next iteration **not found** will be false and the while condition will be false, regardless of the value of **more_to_search**. At line 8, i is incremented by 1 to examine the next element in the list. We need to check if the list is exhausted at line 9, making sure we have more elements to search. That is, when i becomes equal to or larger than the length of the list, the search must stop. We have also replaced the print statement with a return statement, giving us a cleaner function definition.

The sorting problem: The Sorting problem is specified as follows:
Input: a list of *n* elements [a_0, a_1, a_2, … , a_{n-1}]
Output: The same list sorted in ascending order

EXAMPLE 17 – Sorting problem

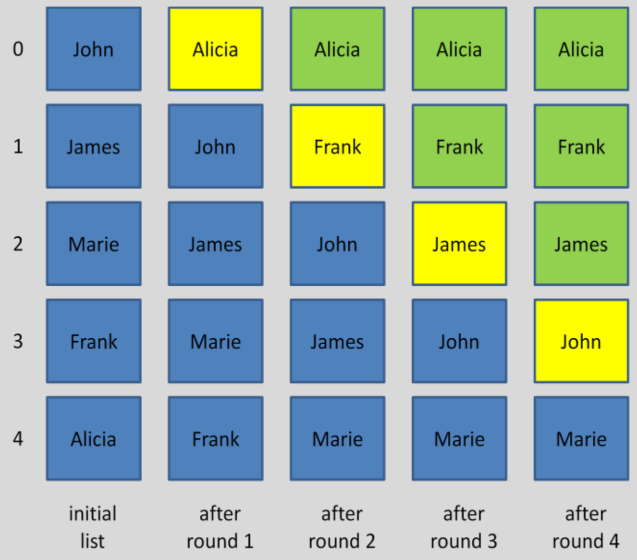
Input: [12, 5, 6, 100, 3, 1]
Output: [1, 3, 5, 6, 12, 100]

Input: [James, Frank, Alicia, Bob, Alice]
Output: [Alice, Alicia, Bob, James]

BubbleSort algorithm: Given a list of *n* elements, BubbleSort sorts the list in *n-1* rounds, or iterations. In round 1, the smallest element in the list "bubbles" up to position 0, where it belongs. In round 2, the second smallest element in the list "bubbles" up to position 1. In general in round i, the ith smallest element in the list "bubbles" up to position i-1.

EXAMPLE 18 – BubbleSort rounds

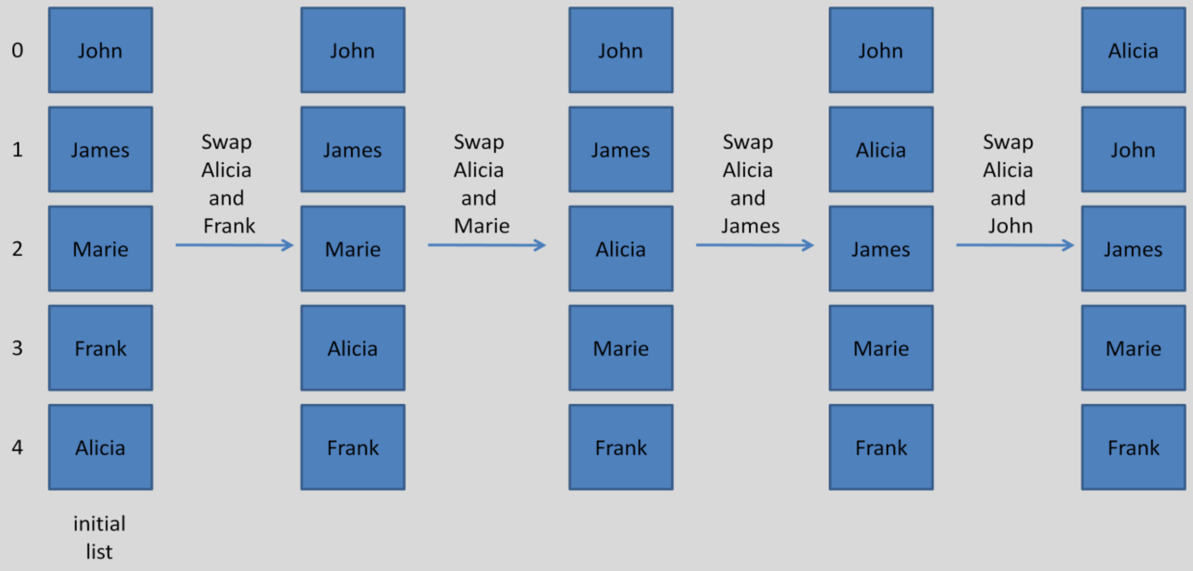
The following visualizes BubbleSort's rounds on a list of 5 elements. In the first round, Alicia, the smallest element bubbles up to position 0. In round 2, Frank bubbles up to position 1. In round 3, James bubbles up to position 2. In round 4, John bubbles up to position 3.



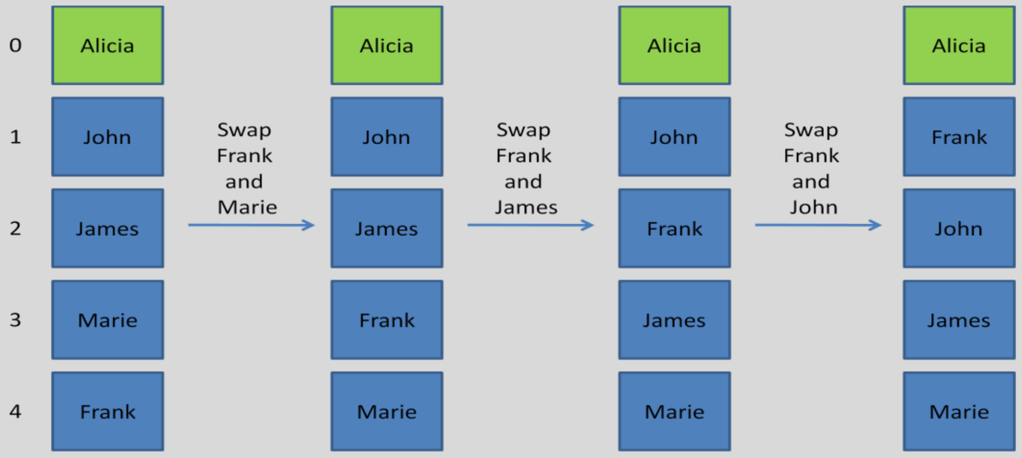
Of course, the crux of the algorithm is how the "bubbling" in each round is taking place. In round *i* starting at the last index in the list, *n-1*, BubbleSort compares the element at *n-1* with the element at *n-2*. If list[*n-1*] < list[*n-2*], then the elements are swapped. Now the elements at index *n-2* is compared with the one at *n-3* to see if a swap is needed. This is repeated until the *i*th smallest element is at index *i-1*.

EXAMPLE 19 – BubbleSort

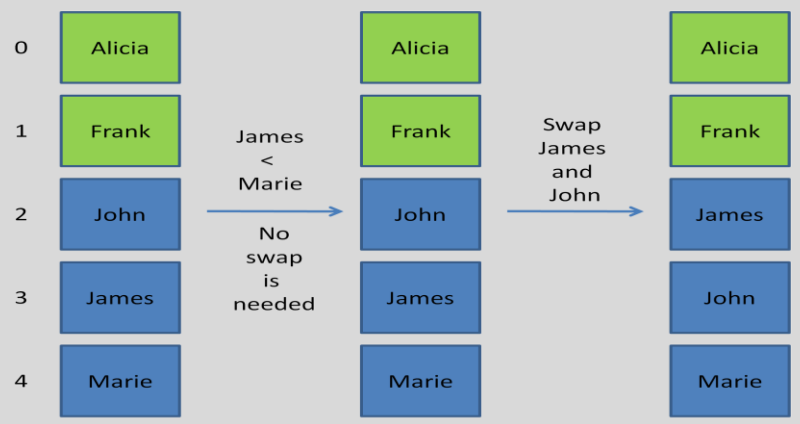
In the first round, Alicia bubbles up to index 0.



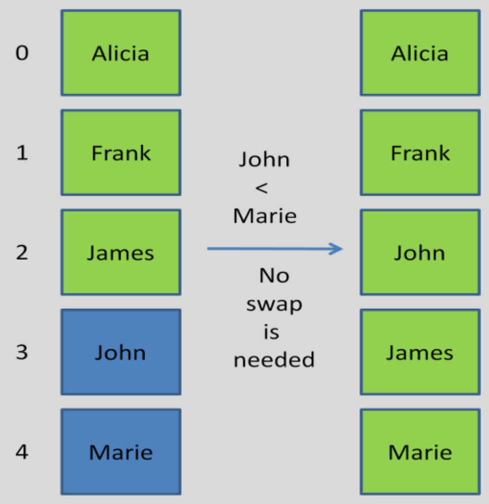
In the second round, Frank bubbles up to index 1



In the third round, James bubbles up to index 2



In the fourth round, John bubbles up to index 3



The list is sorted.

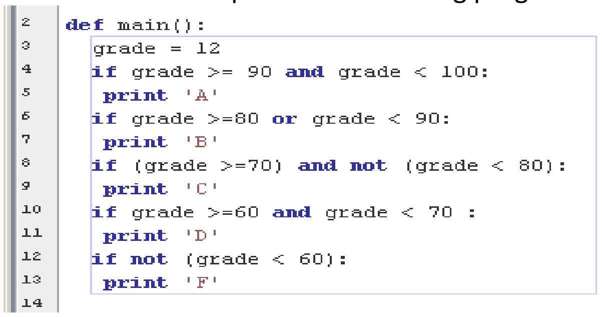
The BubbleSort algorithm is specified in Jython as follows:

```
1 def bubble_sort(list):
2   size = len(list)
3   for i in range(0,size-1):
4     for j in range(size-1,i,-1):
5       if list[j] < list[j-1]:
6         temp = list[j]
7         list[j] = list[j-1]
8         list[j-1] = temp
9   print list
```

Lines 6 to 8 swap the contents of list[j] and list[j-1]. This is done only if need be (list[j] > list[j-1]) , enforced by the if statement at line 5. The outer loop (line 3) performs size-1 rounds. The inner loop (line 4) bubbles up an element to the appropriate position. Since this appropriate position depends on the round number (ith smallest element bubbles up to position i-1 in round i), the inner loop depends on the round number i. Bubbling up starts at position size-1 and ends at position i+1, captured by the function **range(size – 1, i, -1)**.

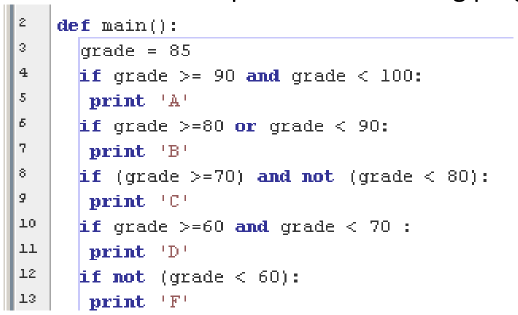
Exercises

1. Write a Jython function that accepts three arguments and returns their average.
2. What is the output of the following program?



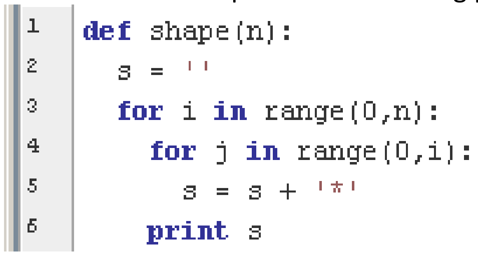
```
def main():
  grade = 12
  if grade >= 90 and grade < 100:
   print 'A'
  if grade >=80 or grade < 90:
   print 'B'
  if (grade >=70) and not (grade < 80):
   print 'C'
  if grade >=60 and grade < 70 :
   print 'D'
  if not (grade < 60):
   print 'F'
```

3. What is the output of the following program?



```
def main():
  grade = 85
  if grade >= 90 and grade < 100:
   print 'A'
  if grade >=80 or grade < 90:
   print 'B'
  if (grade >=70) and not (grade < 80):
   print 'C'
  if grade >=60 and grade < 70 :
   print 'D'
  if not (grade < 60):
   print 'F'
```

4. Write a Jython function that returns the XOR of two Boolean values passed as parameters
5. What is the output of the following program



```
def shape(n):
  s = ''
  for i in range(0,n):
    for j in range(0,i):
      s = s + '*'
    print s
```

6. Look for the insertion sort algorithm, understand it, and write it in Jython

Chapter 3
Organized Thought: Graphs and Trees

A computational problem is a specification of the relationship between the input and output. Input is likely to be some data that the computer needs to transform to an appropriate output. Such input must be prepared and put in an appropriate format. *Graphs* can represent a large class of problems. *Trees* are a special (but very handy) case of graphs.

While the problem specifies the *what*, computers need specific step-by-step instructions to know *how* to convert some input to some output. An *algorithm* is a specification of such *how*. The preparation of an input to an appropriate format makes the *how* easier, if the input is represented properly. Algorithms can be specified in different ways: from pseudo-code to programs. Some algorithms can be specified using finite state machines (abbreviated FSM).

This chapter introduces graphs, FSMs, and trees.

I – Graphs

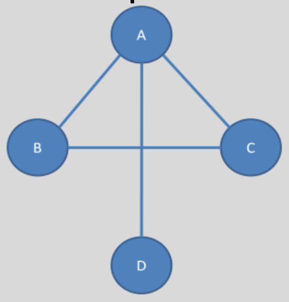
A *graph* is simply a graphical representation of a relation defined on a set. It has a set of elements, called *vertices*, and a set of *edges* that connect these vertices. Formally a graph is a pair of sets, G = (V,E), where V is a set of vertices and E $\subseteq$ V×V (that is, E is a relation on V).

If the edges are *one-way*, the graph is called a *directed graph.* If the edges are *two-way*, the graph is *undirected*. It is sometimes also possible to label edges, the resulting graph is called a *labeled graph*, or a *network*.

EXAMPLE 1 – Undirected graph

The following graph has V = {A, B, C, D} and E = {{A,B},{A,C},{A,D},{B,C}}. The vertices are drawn as circles and the edges are two-way (there is no indication on the edge that forces it to flow in one direction rather than the other). Note that we have used set notation to indicate undirected edges. So, {A,B} is the same edge as {B,A}.

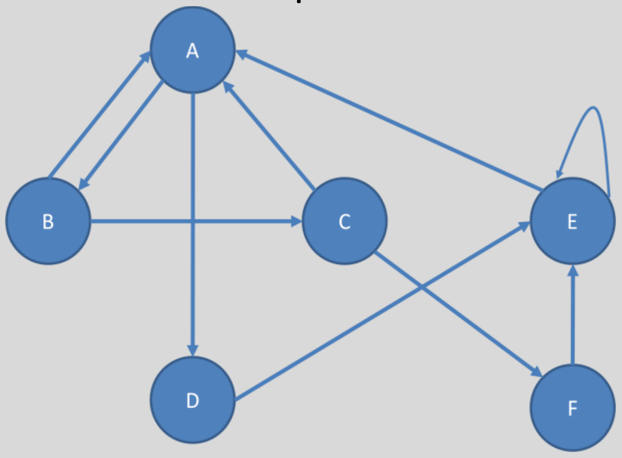
Graph G1



EXAMPLE 2 – Directed graph

The following graph has V = {A, B, C, D, E, F} and the set of edges Ed = {(A,B), (B,A), (B,C), (A,D), (C,A), (D,E), (C,F), (E,A), (F,A), (E,E)}. The edges are one-way, indicated by arrows. Note that in the set E, we used order pairs to indicate one-way edges. (A,B) indicates that there is an edge from A to B, but there is also the pair (B,A) indicating another one-way edge from B to A. Note also the edge (E,E) from E to itself.

Graph G2



Undirected graph terminology: Two vertices in an undirected graph are called *adjacent* if they are connected by an edge. In graph G1, A is adjacent to B, C, and D; however, C and D are not adjacent. A *(undirected) path* is a list of vertices v_1, v_2, …, v_n, such that v_1 is adjacent with v_2 and v_2 is adjacent with v_3 and so on. In G1, there is a path from D to C: D, A, B, C. Notice that this is not the only path from D to C. A path is called a *cycle* if $v_1 = v_n$, or in other words it starts and ends at the same vertex. A cycle is also called a *circuit*. The following path in G1 is a cycle: A, B, C, A. The *degree* of a vertex is the number of adjacent vertices it has. So the degree of A in G1 is 3 since it has three adjacent vertices. The degree of B is 2 and that of D is 1.

Directed graph terminology: If there is a directed edge from vertex v_1 to v_2, v_1 is said to be *adjacent to* v_2, or alternatively v_2 is *adjacent from* v_1. In G2, A is adjacent to D and E is adjacent from D. A *(directed) path* is a list of vertices v_1, v_2, …, v_n, such that v_1 is adjacent *to* v_2 and v_2 is adjacent to v_3 and so on. A cycle (or circuit) is defined similarly to undirected graphs. The path in G2: B, C, A, D, E, B is a cycle. The *in-degree* of a vertex is the number of vertices that are adjacent to it and the *out-degree* is the number of vertices that are adjacent from it. In G2, A has an in-degree of 3 (3 arrows are coming in) and out-degree of 2 (2 arrows are going out).

EXAMPLE 3 – The WWW

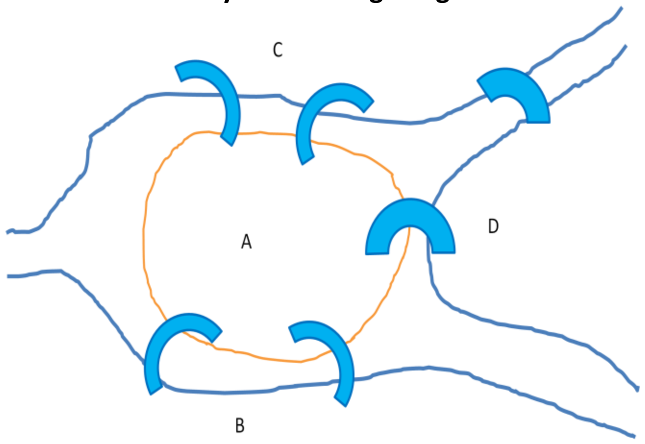
The Web can be modeled as a directed graph. The vertices are the Web pages and the edges are the hyperlinks.

EXAMPLE 4 – Flowcharts

Flow charts can be regarded as labeled directed graphs with two types of vertices: *normal* and *decision* vertices. The labels can be chosen from the set {*none, true, false}*.

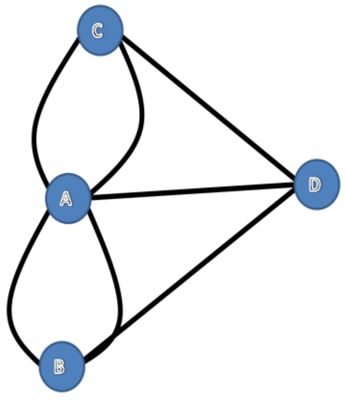
Euler tour and circuit: The town of Konigsberg has two rivers passing through it with seven bridges.

Layout of Konigsberg



Land is marked with A (island), B, C, and D. The question is if there exists a tour that starts somewhere in town, crosses all bridges, crossing each bridge exactly once, and ending up where the tour started. To answer the question, a graph model of the town is first built.

Graph Model of Kongsberg



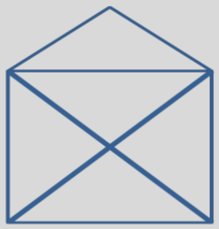
The vertices are land components and the edges are the bridges connecting them. So the question amounts to finding a circuit that crosses all edges and crossing each edge once. Such a circuit is called *Euler* circuit. What Leonard Euler (18th century) noticed is that if a graph has at least one vertex with an odd degree, then a Euler circuit does not exist. Such an odd vertex, call it O, cannot be the starting and ending point of the tour. To start from O, an edge must be used to leave and a different edge must be used to end the tour at O. So, O has to have a degree of at least 2, which is not odd. Well what if you come back to O during the tour to leave again, before the tour ends. In such a case, every time you visit O, you must be coming in using a new edge (never used before) and you must leave it using another new edge. Hence, each such visit uses 2 edges, leaving us with an even degree for the start vertex. The vertex O cannot be the start/end vertex, but cannot also be any other vertex. Any other vertex would also have an even degree because you always visit (it is not the start vertex) this vertex using a new edge and you must leave it (it is not the end vertex) using another new edge. No matter how many times you visit the vertex, each time you're using an even number of edges and the degree of this vertex must be even, if you do not want to leave uncrossed edges behind.

A simpler version of the Konigsberg problem is to find an *Euler path*. Such a path still needs to cross every bridge once, but can start and end in different places. Euler Noticed that for such a path to exist a graph must have two vertices with odd degrees only. These are the starting and ending vertices.

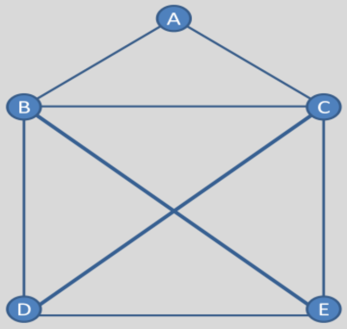
EXAMPLE 5 – Continuous envelope

Can you draw the following envelope shape using the following rules?

- Draw **continuously**, cannot lift the pen from one position to another
- Draw each line **once**, cannot let the pen run on top of an already drawn line



The question reduces to finding an Euler path in the following graph:



This graph has two vertices with odd degrees, namely D and E. So any Euler path must start at one and end at the other. That is you can draw this shape if you start at D and stop at E or vice versa. There is no third way

II – Graph Coloring

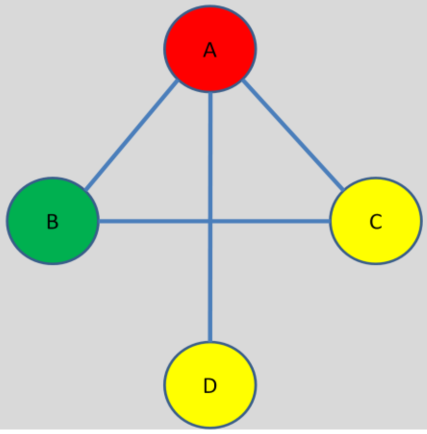
A computational problem specifies the relationship between input and output. One such problem is *graph coloring*, which can be specified as follows:

Input: an undirected graph

Output: a coloring of the vertices of the input graph, such that no two adjacent vertices have the same color and the number of colors is minimized.

EXAMPLE 6 – Graph coloring

Here is a possible coloring for G1. Note that 3 colors is the minimum number of colors that can be used.



One way that comes to mind is to color each vertex with a different color; however, this violates the specifications of the problem, requiring that the number of colors is minimized. This is especially desirable for larger graphs. (Though in some graphs, the minimum number of colors is equal to the number of vertices. Can you think of such a graph?)

So how do we transform the input to the required output? This is the job of the *algorithm*. That is an algorithm solves a computational problem. There are different ways by which an algorithm is specified. The most common one is to use *pseudo-code*, something that resembles a computer program, yet it is mainly written in English. Pseudo-code allows us to specify an algorithm in a general way that does not depend on a specific programming language format. Of course an algorithm can be also specified using actual code, with a specific programming language. Sometimes, visual representation of algorithms using flow-charts can be very useful, especially to beginners. Flowcharts themselves can be regarded as directed graphs. There are some algorithms that can be represented in a special labeled directed graph format, called finite state machines (abbreviated FSM). We will study FSM later in this chapter.

Graph coloring algorithm:

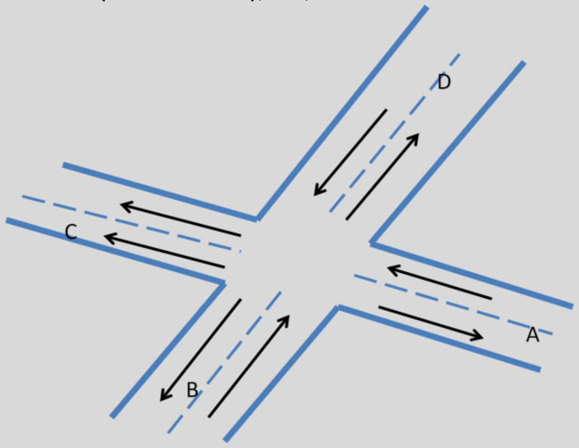
Repeat the following two steps until all the vertices in the input graph are colored

1. Select an uncolored vertex and color it with a ***new*** color, **C**
2. For each uncolored vertex,
    - Determine if it has an edge with a vertex that is colored with color **C**
    - If not, color it with color **C**
    - If yes, skip it

Graph coloring can be very useful in scheduling problems. The items that need to be scheduled are arranged into a graph as vertices, and the edges indicate which items are conflicting, cannot be scheduled at the same time. Coloring the graph allows us to group the items so that the ones in the same group (have the same color) can be scheduled at the same time.

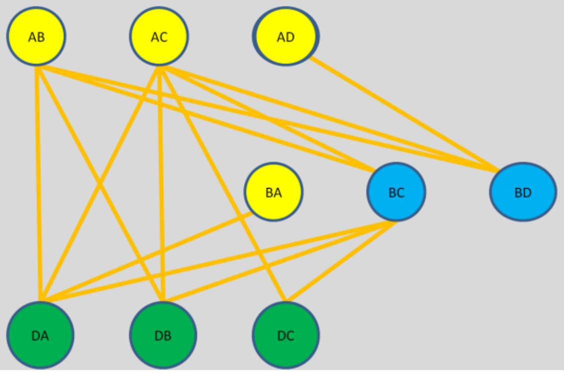
EXAMPLE 7 – Conflicting turns

Consider the following traffic intersection. The objective is to determine which traffic flows can be run simultaneously. Intersections are normally an expensive resource and maximizing their use is desirable. The naïve solution of letting traffic flow in one direction is undesirable; after all, we do not do this in real life. For instance, traffic from A to D (denoted AD), AC, and BA can all run simultaneously.



This is a scheduling problem and we can use graph coloring to determine the groups of flows that are non-conflicting. Since the graph coloring algorithm minimizes the number of colors used, it maximizes the traffic flow.

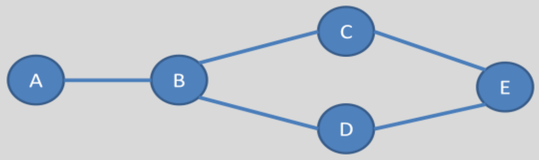
The following is graph model where each vertex represents a traffic flow direction. These that conflict are connected by edges. The graph is also colored using the graph coloring algorithm. It shows that there are three groups of traffic flow directions that can be run at the same time.



Adjacency matrix: How is the graph input to a computer? One famous format is an *adjacency matrix*, which is simply a table that encodes the graph. In the case of unlabeled graphs, the matrix will have values of 0s and 1s only. In such a 0-1 matrix a value of 1 in entry (i,j) indicates the existence of edge (i,j) in the graph.

EXAMPLE 8 – Adjacency matrix (unlabeled graph)

Consider the following graph:



It can be represented using the following adjacency matrix:

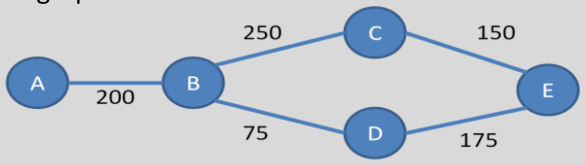
	A	B	C	D	E
A	0	1	0	0	0
B	1	0	1	1	0
C	0	1	0	0	1
D	0	1	0	0	1
E	0	0	1	1	0

A value of 1 in this matrix indicates the existence of an edge. In the first row, only the entry (A,B) is 1, indicating the existence of the edge (A,B). There is no edge between A and C, so the entry (A,C) is 0.

If the graph is labeled, then we can use an adjacency matrix where the 1's are replaced by the labels of the corresponding edges

EXAMPLE 9 – Adjacency matrix (labeled graph)

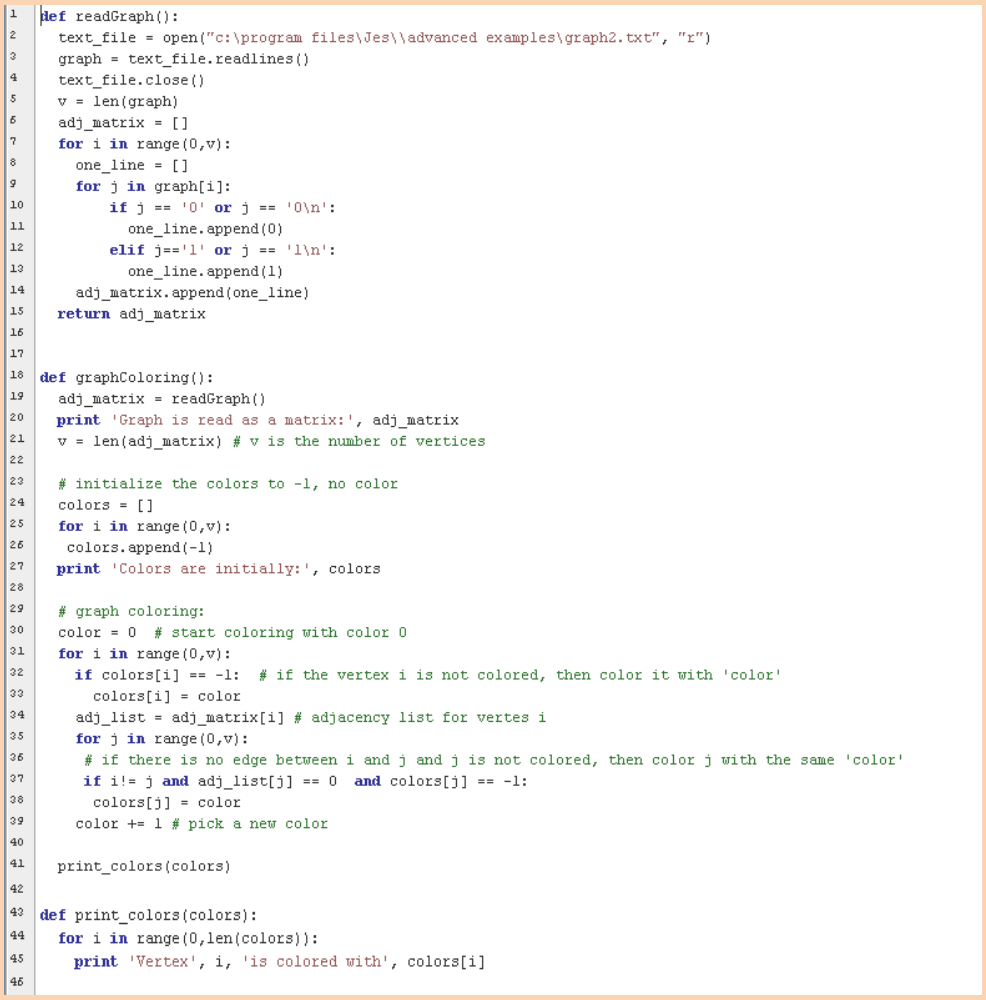
Consider the following labeled graph:



It can be represented using the following adjacency matrix:

	A	B	C	D	E
A	0	200	0	0	0
B	200	0	250	75	0
C	0	250	0	0	150
D	0	75	0	0	175
E	0	0	150	175	0

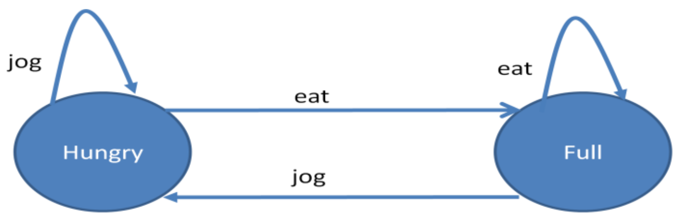
Advanced Topic – Graph Coloring in Jython



```
def readGraph():
  text_file = open("c:\program files\Jes\\advanced examples\graph2.txt", "r")
  graph = text_file.readlines()
  text_file.close()
  v = len(graph)
  adj_matrix = []
  for i in range(0,v):
    one_line = []
    for j in graph[i]:
        if j == '0' or j == '0\n':
          one_line.append(0)
        elif j=='1' or j == '1\n':
          one_line.append(1)
    adj_matrix.append(one_line)
  return adj_matrix


def graphColoring():
  adj_matrix = readGraph()
  print 'Graph is read as a matrix:', adj_matrix
  v = len(adj_matrix) # v is the number of vertices

  # initialize the colors to -1, no color
  colors = []
  for i in range(0,v):
   colors.append(-1)
  print 'Colors are initially:', colors

  # graph coloring:
  color = 0  # start coloring with color 0
  for i in range(0,v):
    if colors[i] == -1:  # if the vertex i is not colored, then color it with 'color'
      colors[i] = color
    adj_list = adj_matrix[i] # adjacency list for vertes i
    for j in range(0,v):
     # if there is no edge between i and j and j is not colored, then color j with the same 'color'
     if i!= j and adj_list[j] == 0  and colors[j] == -1:
      colors[j] = color
    color += 1 # pick a new color

  print_colors(colors)

def print_colors(colors):
  for i in range(0,len(colors)):
    print 'Vertex', i, 'is colored with', colors[i]
```

III – Finite State Machines

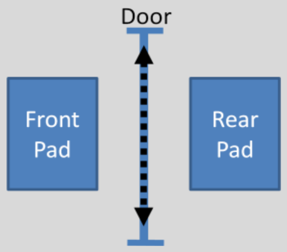
A finite state machine is a special kind of a directed graph with two properties: the set of vertices, called *states*, is finite and the edges are *labeled transitions*. The labels are typically called events. FSMs are very handy when designing hardware algorithms (controllers) as we will see shortly. Consider the following FSM:



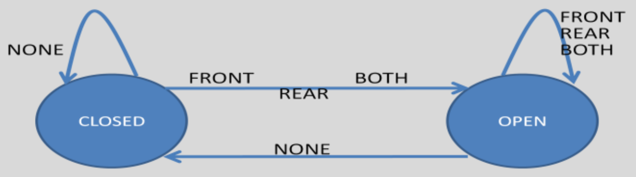
This FSM has two states: *hungry* and *eat*. The directed edges are labeled with *events* that trigger transitions from one state to another. So, if you are in state *hungry* and you jog, you will still be *hungry*. However, if you eat, the state changes to *full*. Eating while you're *full* does not change the fact that you're *full*, but jogging will make you *hungry* again.

EXAMPLE 10 – FSM for a sliding door controller

Consider an automatic sliding door that opens when people step on the front or rear pads



An FSM that controls the door is as follows:



The events that can occur are:

- FRONT: someone steps on the front pad
- REAR: someone steps on the read pad
- BOTH: both pads have someone stepping on them
- NONE: neither pad has someone stepping on it

We allow for readability an edge to have more than one label. In fact, this is an abbreviation for having many edges between two states with different labels. So, if the door is CLOSED and any of the events FRONT, REAR, or BOTH takes place, the door should OPEN. It only goes back to the CLOSED state in the case of the NONE event.

Of course, we can represent the FSM as a matrix, since it is just a graph, as follows:

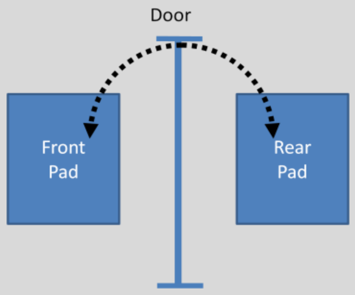
Events

Door States	NONE	FRONT	REAR	BOTH
CLOSED	CLOSED	OPEN	OPEN	OPEN
OPEN	CLOSED	OPEN	OPEN	OPEN

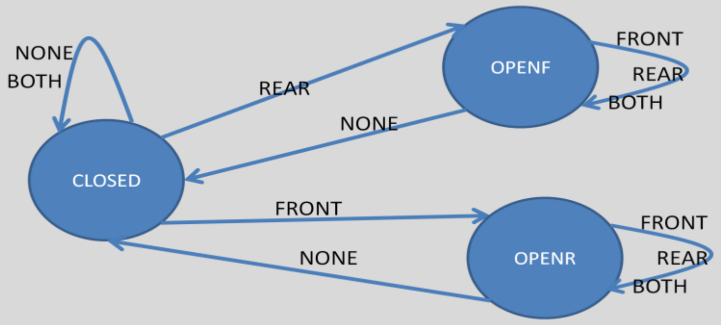
The adjacency matrix corresponding to an FSM is often called a *state table* and the FSM graph is also called a *state diagram*.

EXAMPLE 11 – FSM for a two-way opening door controller

This door does not slide, it opens both ways towards the pads



An FSM that controls the door is as follows:



In this version, we have to split the OPEN state to two: OPENR, open to the rear, and OPENF, open to the front. This is necessary because if someone steps on the front pad (respectively, rear pad) it will be unsafe to open the door to the front (respectively, the rear).

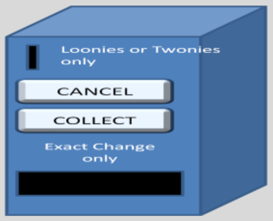
The FSM matrix is as follows:

Events

Door States	NONE	FRONT	REAR	BOTH
CLOSED	CLOSED	OPENR	OPENF	CLOSED
OPENF	CLOSED	OPENF	OPENF	OPENF
OPENR	CLOSED	OPENR	OPENR	OPENR

EXAMPLE 12 – FSM for a simple vending machine

A simple vending machine dispenses $3 phone cards. It accepts $1 and $2 coins only. It does not give any change. If the user exceeds the $3 amount, the machine gives the money back without dispensing any phone cards. It can dispense one phone card at a time. The machine has two buttons: CANCEL to cancel the transaction and COLLECT to dispense the phone card.



The machine keeps the collected coins in a *coin collector*, until a card is dispensed, in which case the coins are dropped into the piggy bank and cannot be recovered by the user.

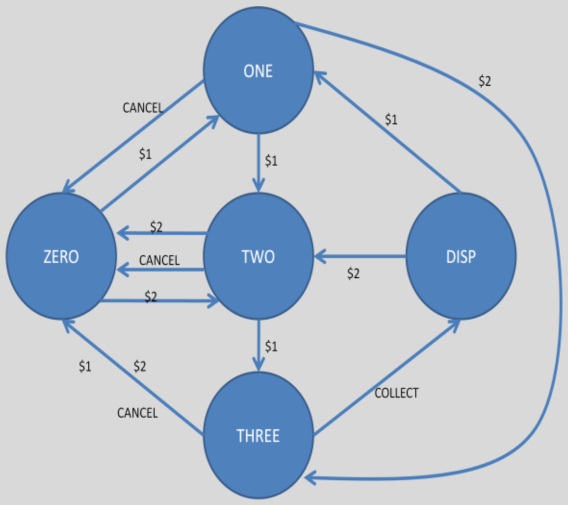
For this FSM, we need five states:

- ONE: Total in coin collector is $1
- TWO: Total in coin collector is $2
- THREE: Total in coin collector is $3
- DISP: dispenses a card, roll in coins to piggy (coin collector becomes empty)
- ZERO: return the coins in the coin collector; also serves as a start state (Initially the coin collector is empty)

The events are:

- $1: user inserts a loonie
- $2: user inserts twonie
- CANCEL: user presses CANCEL
- COLLECT: user presses COLLECT

The FSM is:



IV – Trees

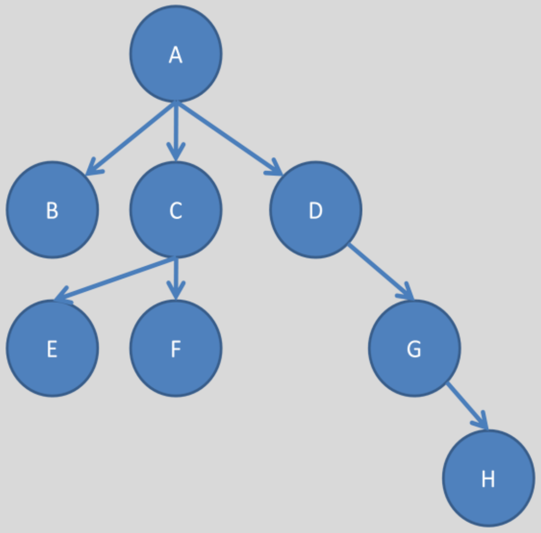
Trees are a special case of graphs. A *tree* is a (directed) graph with the properties:

- There is a designated vertex, called the *root* of the tree
- There is a unique directed path from the root to every other vertex in the tree

EXAMPLE 13 – Trees

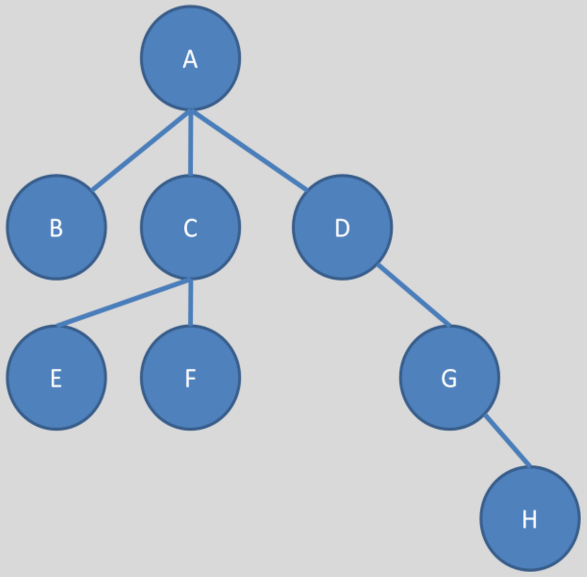
A tree is drawn downwards, with the root on the top. The root of the following tree is A.

Tree T



Often, we drop the arrows from the edges, since it is always understood that the edges direction is downwards. The following is the same tree:

Same Tree T

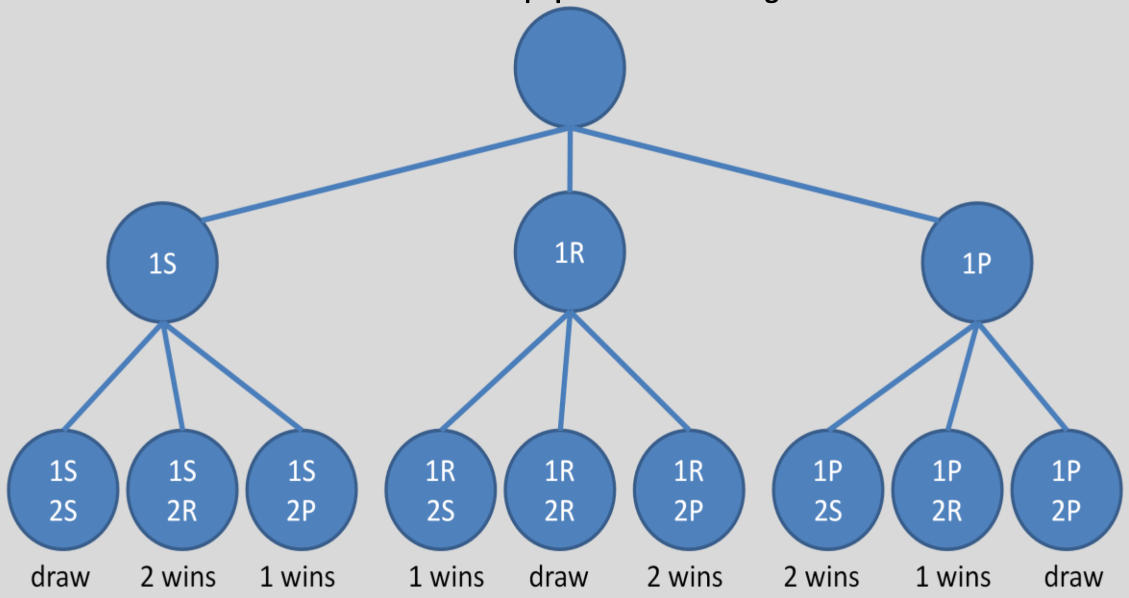


Tree terminology: A vertex which is adjacent from some vertex V, is call a *child* of V. In tree T, E and F are children of C. If a vertex V is a child of vertex P, then P is the *parent* of V. Note that a vertex can have at most one parent. All vertices have exactly one parent, except the root which does not have a parent. A vertex that does not have children is called a *leaf*. In tree T, the vertices B, E, F and H are leaves. An *ancestor* of a vertex V is either the parent of V or an ancestor of the parent of V. A *descendant* of vertex V is either a child of V or a descendent of a child of V. The *distance* of a vertex V from the root is the number of edges on the path from the root to V. In tree T, B is at distance 1 from the root, and H is at distance 3 from the root. The vertices that are at the same distance from the root are called *siblings*.

EXAMPLE 14 – Game trees

In games, trees explore all possibilities of a game. The root is the start of the game and the path from the root to a leaf explores one possible outcome of the game.

Game tree for the paper-rock-scissors game



In this tree, 1S indicates that player 1 chooses scissors; 1R indicates that player 1 chooses rock; and 1P is player's 1 choice of paper. The notation is similar to player 2. Each path from the root to a leaf captures on possible instance of the game.

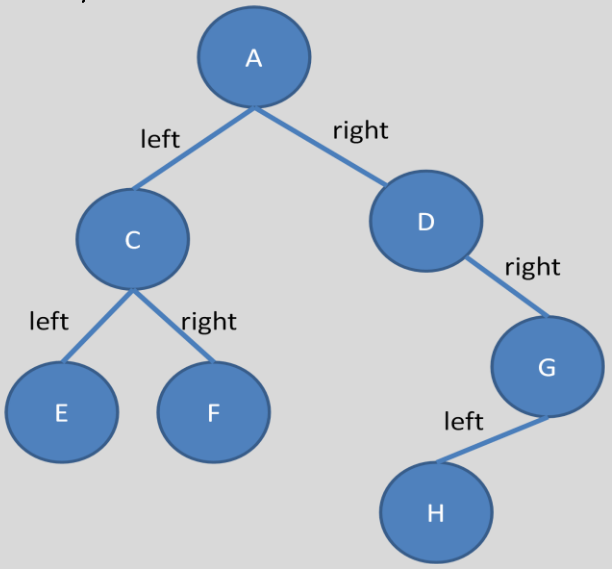
For other more complex games, the tree can become prohibitively large such as in the game of chess.

Binary trees: A binary tree is a tree with the following properties:

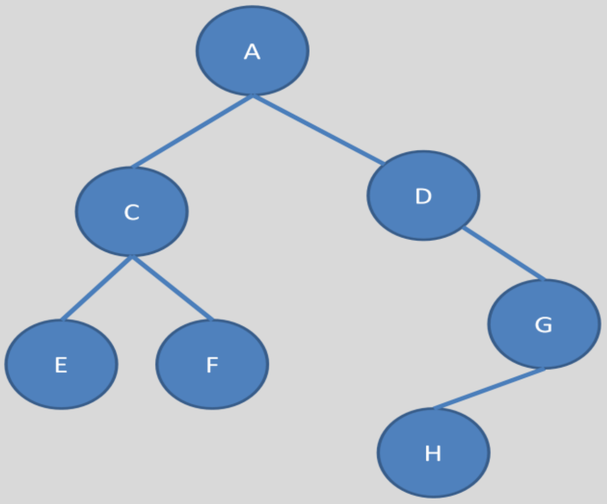
- The edges are labeled with the labels *left* or *right*
- Every vertex has at most two children; if both children exist, then one edge must be labeled with *left* and the other *right*.

EXAMPLE 15 – Binary trees

The following is an example binary tree.



Just as we have dropped the arrows, we can also drop the labels, if we always draw the tree in such a way that left children are drawn on our left and the right ones on our right. The above, binary tree can be drawn as follows, without any confusion about who's who in the left and right world (and we are not talking politics here!)



V – Huffman's Coding

Everything in the computer is represented as 0s and 1s, including text and multimedia files. A text file is encoded using 0s and 1s with predefined fixed-length codes. One famous way of encoding is ASCII (American Standard Code for Information Exchange) codes. ASCII gives each symbol a code of 0s and 1s. For instance:

- ASCII code for character A is 0100 0001
- ASCII code for B is 0100 0010
- ASCII code for C is 0100 0011
- ASCII code for D is 0100 0100

The ASCII codes are of length 8. Everything you can type on the keyboard has an ASCI representation, including space, escape, and control.

EXAMPLE 16 – ASCII coding

The word ACE is represented in ASCII by 010000010100001101000101. To know what a code really represents, read the first 8 digits and find out what they represents, then read the next 8 and henceforth.

ASCII codes of length 8 can represent 2^8 different characters, which is 256 characters. In general, if you're coding characters with codes of length **n** and the coding uses **a** symbols, you can represent $\mathbf{a}^{\mathbf{n}}$ different characters. (In the case of ASCII **a** = 2 because the codes consist of two symbols 0 and 1, and **n** = 8 because the length of the codes are 8.) If the symbols are 0s and 1s, they are called *bits.*

EXAMPLE 17 – Coding

Assume that we have a file that contains strings formed out of 6 characters only: A, I, C, D, E, and S (for space). If such a file has 100 characters, how many bits are needed to code a file? First we need to determine the length of the codes. One attempt is to use codes of length 2, but this is not enough. This would only allow me to represent 4 distinct characters, since $2^2 = 4$. Since $2^3 = 8$, 3 bits is more than enough because we have 6 characters only. Therefore, the file would require 3 x 1000 = 3000 bits.

Compression: So how does compression of files work? The idea is that if some characters are more frequent than others, we should give them shorter codes. This gives rise to *variable-length codes*.

EXAMPLE 18 – Coding

Assume that we have analyzed the file of Example 17 and found the following statistics:

- 35% of the characters in the file are S
- 28% are A
- 20% are E
- 7% are I
- 6% are C
- 4% are D

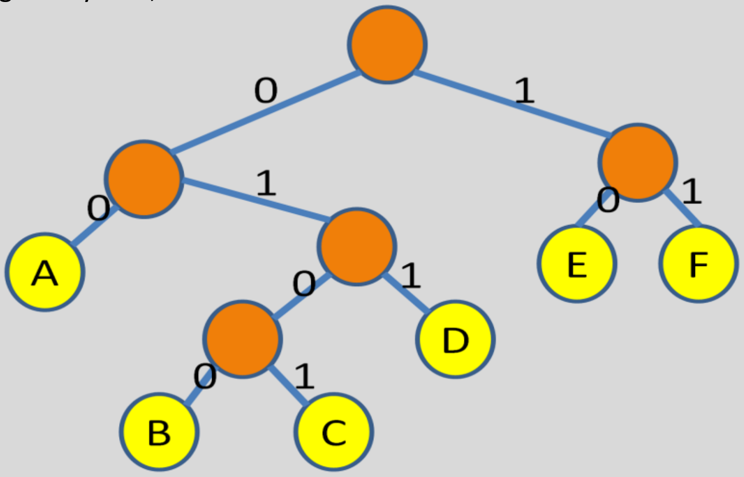
If we use 2 bits to represent each of S,E, and A (the most frequent ones), 3 bits to represent I and 4 bits for each of C and D, this should result in a coding that requires 24% fewer bits than 3000.

If we use this coding, what is the size of the file? 350 S's (35% of 1000) require **700** bits (2 bits for each S); 200 E's require **400** bits; 280 A's require **560** bits; 70 I's require **210** bits; 60 C's require **240** bits; 40 D's require **160** bits; Total is **2270** bits; Recall that with fixed codes, the size is **3000** bits. The compressed file size is about **76%** of the original file size**.**

Non-prefix codes: Not any variable-length codes work. Assume A's code is 0; C's code is 1; and E's code is 01. The code 0101 could correspond to ACE, EAC, ACAC, or EE. Codes that work must have the property that each code corresponds to exactly one value (it has a unique interpretation). They must have the property: No code can be the prefix of another code. These are called *non-prefix* codes. **0** is a prefix of **0**1, this is why our coding failed. Non-Prefix codes can be generated using a binary tree.

EXAMPLE 19 – Coding

Consider the following binary tree, where the characters are all in leaf vertices:



The left edges are labeled with 0 and right edges with 1. Recording the labels on the path from the root to the leaves, each path corresponds to a non-prefix code. The codes are: A's code: 00; B's 0100; C's 0101; D's 011; E's 10; and F's 11.

The code 111010011 represents the word FEED and there is no confusion about that.

So how do we generate non-prefix codes such that the most frequent characters have shorter codes than the least frequent characters? David Huffman's algorithm allows us to do just this.

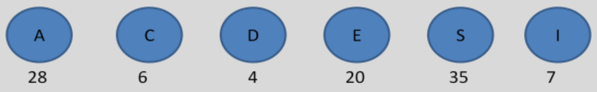
Variable-length coding problem: This problem is specified as follows
Input: a list of characters and their frequencies
Output: Non-prefix variable-length codes for each input character, such that the characters with higher frequencies have shorter codes, where possible

Huffman's coding algorithm:

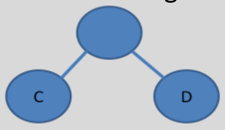
1. Assign to each symbol its weight (frequency); each of these symbols represent a tree of one vertex only; call this collection of trees a *forest*
2. Repeat step 3 until we have a single tree in the forest
3. Choose two trees that have the minimum weights
   i. Replace these two trees with a new tree with new root
   ii. Make the tree with the smaller weight a right child
   iii. The weight of the new tree is the sum of old weights
4. Label the left edges with 0s and the right edges with 1s. Assign each leaf the labels of the edges on the path from the root to that leaf.

EXAMPLE 20 – Huffman's coding

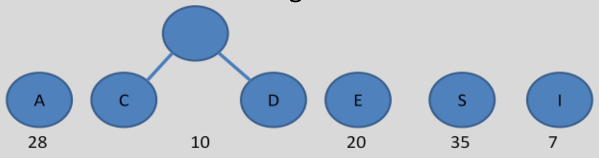
Let the input be: (A,28%),(C,6%),(D,4%),E(20%),(S,35%),(I,7%). Step1 of the algorithm results in the following forest:



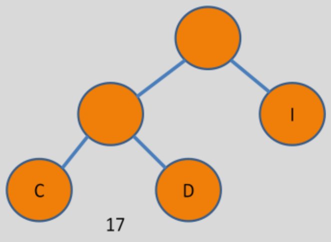
There are 6 trees in this forest and each is assigned a weight. Since, we have more than one tree in the forest we perform step 3. The two trees of minimum weights are C and D. We combine them in the tree:



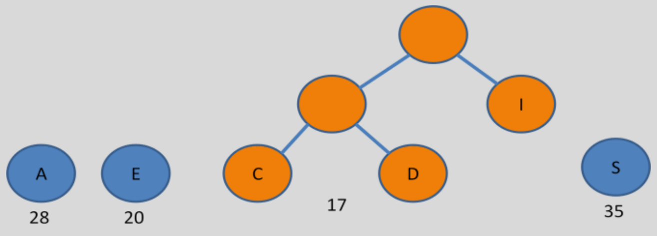
The weight of this new tree is 6+4 = 10. The resulting forest is:



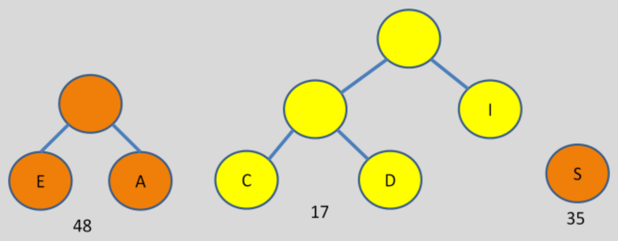
Since we still have more than one tree in the forest, we apply step 3 again choosing the trees of weights 7 and 10 to be replaced by:



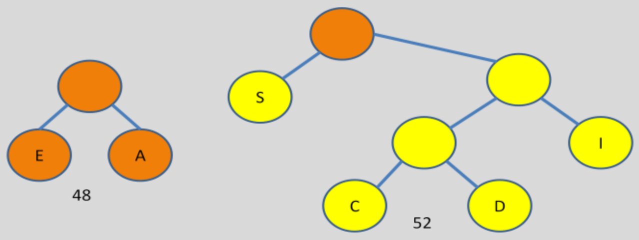
The resulting forest is:



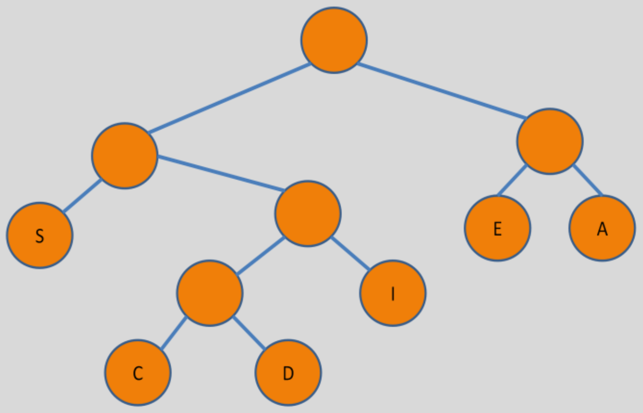
One more application of step 3 results in the forest:



The next candidate trees for step 3 are those with weights 17 and 35. This results in the following forest?



There are only two trees left in the forest, combining these using the last application of step 3 results in the tree:

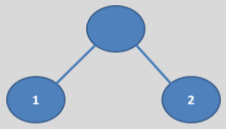


Since the forest has a single tree, step 3 will not be applied again. Step 4 is already demonstrated in Example 14.

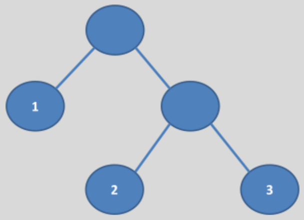
Trees as nested lists: How do we represent trees in the computer? Since trees are graphs, we can use an adjacency matrix. However, this is not the best way. A better way is to use nested lists.

EXAMPLE 21 – Nested lists as trees

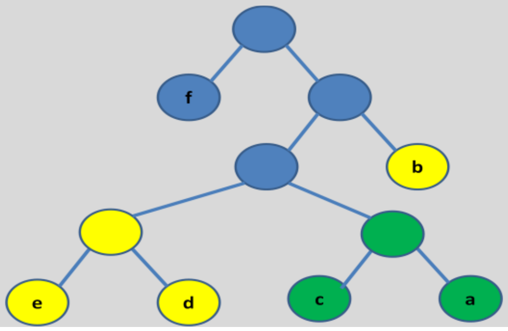
- The list [1,2] represents the tree:



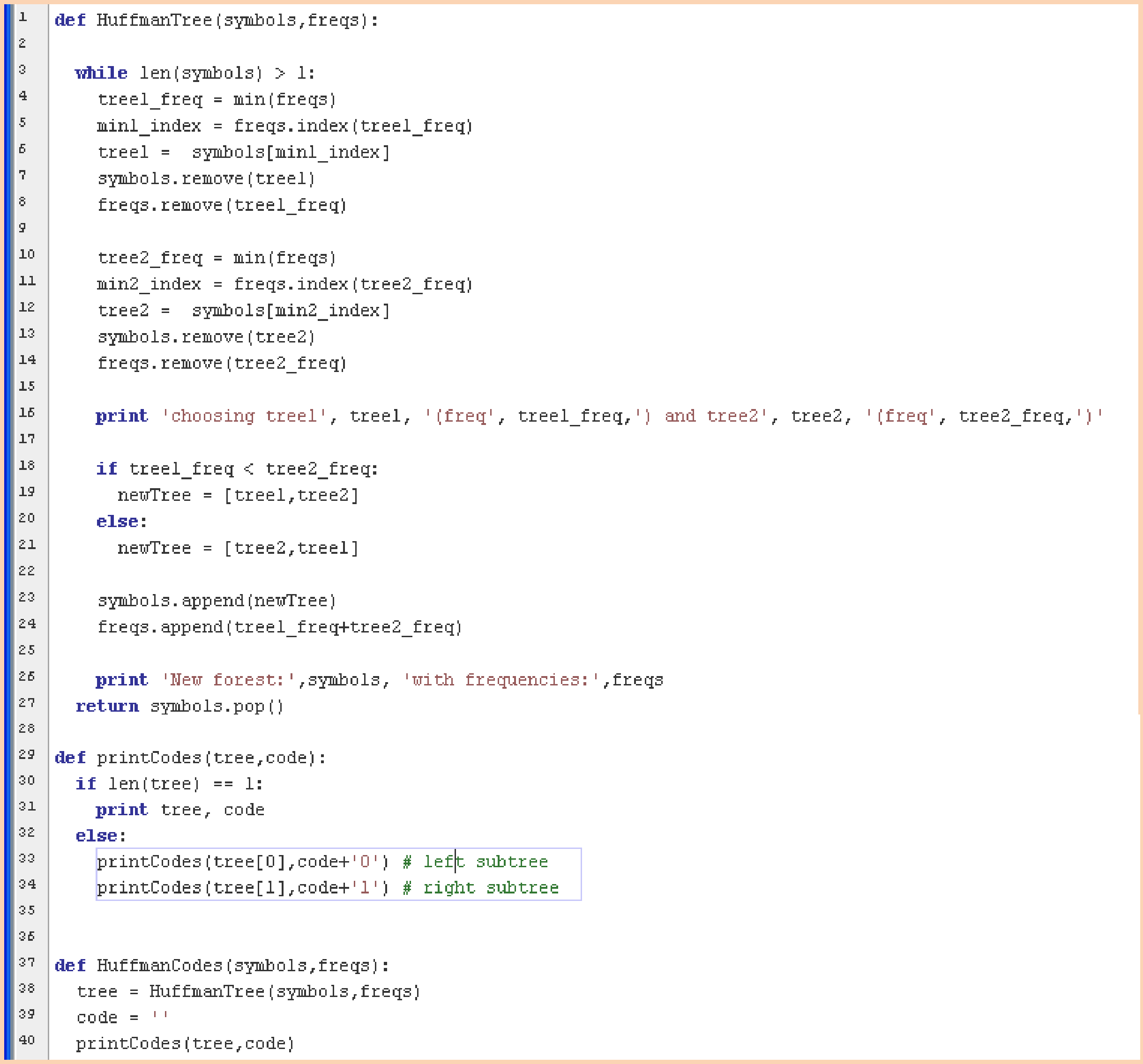
- The list [1,[2,3]] represents the tree:



- The list [f,[[[e,d],[c,a]],b]] represents the tree:



ADVANCED TOPIC – Huffman's coding in Jython



```
def HuffmanTree(symbols,freqs):

  while len(symbols) > 1:
    tree1_freq = min(freqs)
    min1_index = freqs.index(tree1_freq)
    tree1 =  symbols[min1_index]
    symbols.remove(tree1)
    freqs.remove(tree1_freq)

    tree2_freq = min(freqs)
    min2_index = freqs.index(tree2_freq)
    tree2 =  symbols[min2_index]
    symbols.remove(tree2)
    freqs.remove(tree2_freq)

    print 'choosing tree1', tree1, '(freq', tree1_freq,') and tree2', tree2, '(freq', tree2_freq,')'

    if tree1_freq < tree2_freq:
      newTree = [tree1,tree2]
    else:
      newTree = [tree2,tree1]

    symbols.append(newTree)
    freqs.append(tree1_freq+tree2_freq)

    print 'New forest:',symbols, 'with frequencies:',freqs
  return symbols.pop()

def printCodes(tree,code):
  if len(tree) == 1:
    print tree, code
  else:
    printCodes(tree[0],code+'0') # left subtree
    printCodes(tree[1],code+'1') # right subtree


def HuffmanCodes(symbols,freqs):
  tree = HuffmanTree(symbols,freqs)
  code = ''
  printCodes(tree,code)
```

Exercises

1. Identify problems that can be solved by graph coloring.
2. Construct an example graph consisting of at least 5 vertices, such that the minimum number of colors required for the graph is equal to the number of vertices.
3. Design an FSM for a garage door controller. Garage doors receive signals from a remote control with one button. If the button is pressed, the door opens when it is closed and vice versa. If the button is pressed while the door is in motion, the door stops; when the button is pressed again, the door starts moving but it reverses the direction it had before it was stopped.
4. Modify the vending machine FSM to allow the machine to accept two twonies and return a loonie back
5. Design an FSM for a similar vending machine that has two types of cards: $3 and $4 cards. The machine would have two COLLECT buttons to choose one for each type of card. The machine also gives change
6. How many bits are required to code 10 characters (A to H) using fixed length codes. Choose the shortest possible codes.
7. If a file contains 10000 characters from the characters (A to H), how many bits will the size of the file be, using your coding from the previous question.
8. Apply Huffman's algorithm to the following input: (A,23%),(B,60%),(C,4%),(D,3%),(E,2%),(D,2%),(E,2%),(F,2%),(G,1%),(H,1%). Generate variable length codes for the characters A to H.
9. Repeat question 6, using the codes from question 7. What is the percentage saving in the file size from question 6.

Chapter 4
Leavers and Gears: How do Computers Work?

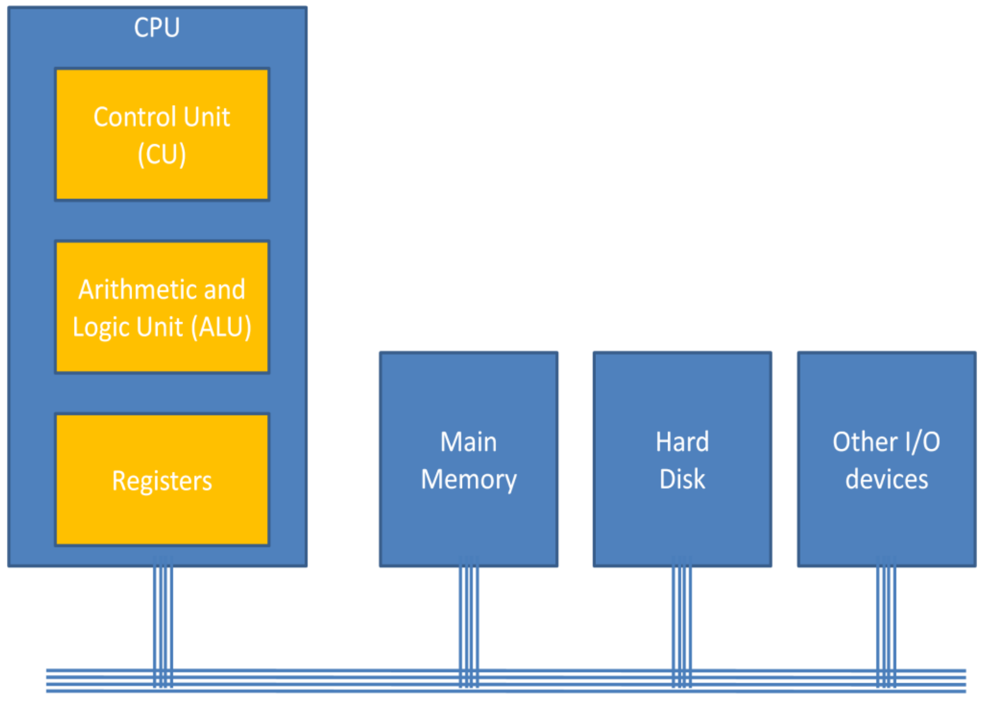
A computer is built as a complex finite state machine connected to memory, where programs and data are stored. The fundamentals of such an FSM are the same as these studied earlier, but the FSM is much more complex. A modern computer consists of a Central Processing Unit (CPU), primary memory, secondary storage, and peripheral devices. We will have a look at each of these components in this chapter.

CPU and memory are typically built using basic building blocks, called logic gates. In fact, the CPU is a complex logic circuit. These gates operate on binary numbers, 0s and 1s, called bits. We will study the binary number system in this chapter as well as logic gates. We will also understand how to build logic circuits from the basic building blocks.

I – The Big Picture

A modern computer consists of a Central Processing Unit (CPU), main memory, secondary memory or storage and peripheral devices.

The organization of a modern computer



The CPU, also called *processor*, is the brain of the computer. It consists of two components: the *Arithmetic and Logic Unit* (ALU) and the *Control Unit* (CU). The ALU performs simple arithmetic and logic operations, such as adding two numbers. The CU controls the operation of the ALU and other components in the machine. CPUs execute programs and while doing so they need a scratch pad memory to store data and intermediate results. Such a scratch pad is provided as a set of *registers*. Each register can hold a data value, such as a number, or a program instruction. A register can hold few bits; the actual size may differ from one machine to another. Nowadays, the register size is typically 32 or 64 bits. The number of registers is quite few, a dozen or so, but the actual number depends on the underlying machine and may vary from one machine to another.

The *main memory* component stores data and programs needed by the CPU. Anything that the CPU operates on must be in main memory. Main memory is typically referred to by RAM, *Random Access Memory*, for historical reasons. Early computers used tapes (just like the out phased music cassettes) for main memory. Tapes are *sequential access memory*; to listen to the last song on the cassette, you would need to fast forward to the appropriate position going through the tape sequentially. There is no way to jump directly to the needed place on the tape, but this is the case with RAM.

Main memory is volatile; it cannot hold any data without a flowing electric current. That is, if you switch the computer off, all what is stored in RAM is lost. Hence, computers need a non-volatile storage, one that holds data in the absence of electric power. There are different non-volatile, or permanent, storage devices. These include *hard disks* and *optical disks*.

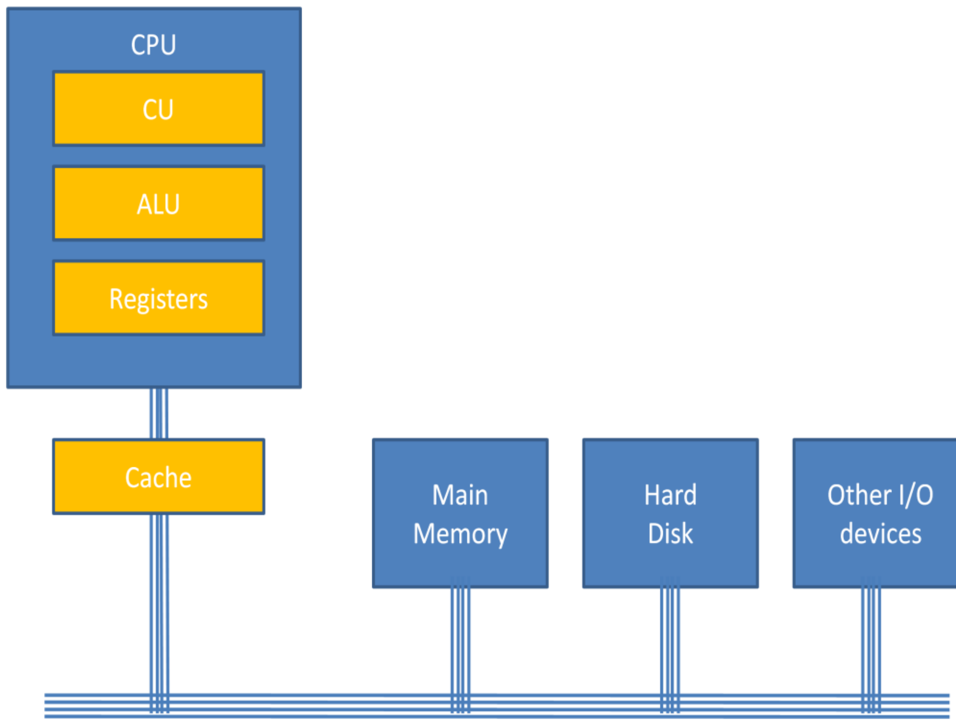
A hard disk makes use of electro-magnetic signals that stay in the absence of power. Optical disks, such as music CDs and DVDs, burn holes into a thin metal layer that can be read by laser beams.

Other peripheral or Input/Output (I/O) devices include a keyboard, screen, mouse, printer, and henceforth.

All of the computer components are connected by a *bus*, a bundle of thin wires each capable of carrying one signal, representing a 0 or a 1.

Main memories are fast, but are slower than CPUs. They typically slow down the CPU's operation. That is why modern machine also include *cache* memory (cache is French for hidden). The cache is faster than main memory but much smaller. A useful analogy here is cooking supplies. If you run out of supplies in the kitchen (the CPU), it would be very costly (time-wise) to run to the supermarket (the main memory) every time you require something. Instead, we typically cache a small amount of supplies in a storage room (the cache), whose access is much faster than making a trip to the supermarket. Of course, the cache can become obsolete (empty in the kitchen supplies analogy) and occasionally you will need to refurbish it from main memory.

The Cache as part of the organization of a computer

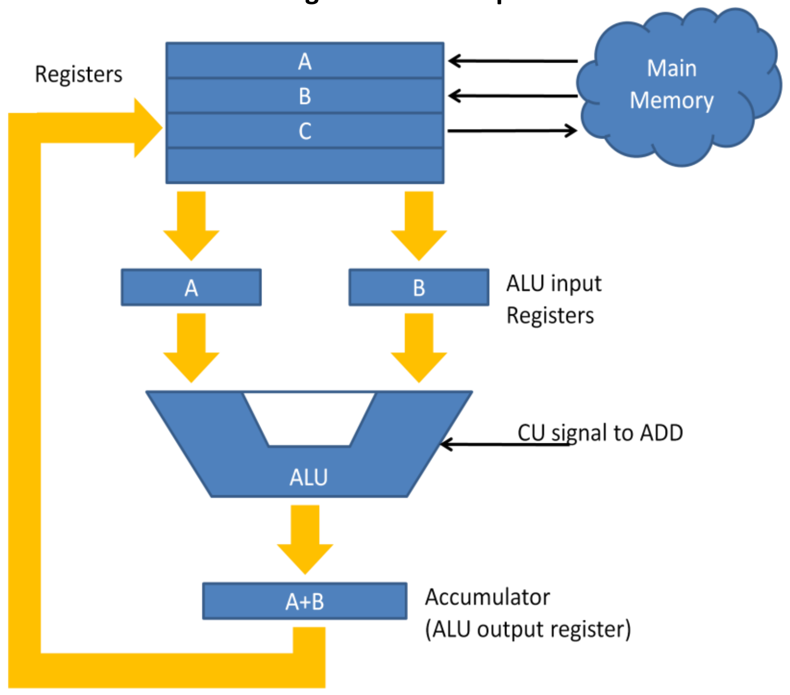


ALU operation: There are handful operations that an ALU can perform: adding two numbers, **and**ing and **or**ing two logic (truth) values, and negating a value. Any other operation, such as multiplication and subtraction, can be performed from these basic ones. For instance, to multiply 5 and 6, the ALU can add 6 to itself 5 times. To subtract 5 from 6, the ALU can negate 5 and add 6 and negative 5.

The basic operation of the ALU is very simple. To perform a basic operation, say add, the following steps must be followed:

Load the first number from main memory to register A
Load the second number from main memory to register B
Load the first input register of the ALU from A
Load the second input register of the ALU from B
Perform an add
Collect the output in the ALU output register (also called the accumulator)
Load the result back to a register C
Store C in main memory

Illustrating an ALU ADD operation



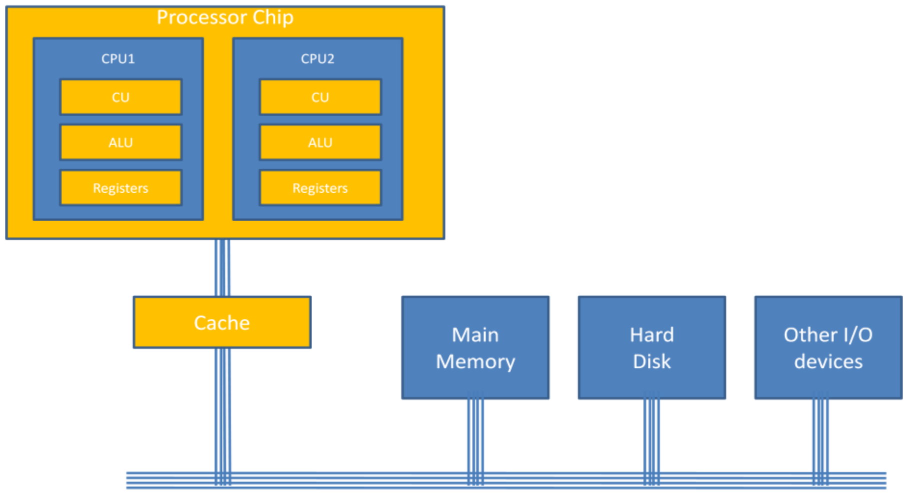
Each of these steps is called a *microinstruction*. A typical computer (at the time of the writing) can perform 3 billion microinstructions per second. This speed of CPUs is measured in Hertz. Such a CPU has a speed of 3 Giga Hertz (GHz).

CPUs have been consistently made faster over the past decades. This was mainly accomplished by jamming more transistors (basic electronic elements that works as switch) into the CPU chip. Now days, it is normal to combine a billion or more transistors into a chip smaller than the size of a credit card. This is resulting in an overheating problem. With so many transistors in a very small area, the chip overheats and this can result in malfunctioning.

Dual core machines: Processor designer resorted to an old alternative idea to make the CPUs faster. If you're building a brick wall, you may hire one mason to do the job. Imagine that this is the fastest mason on earth and it needs him/her 30 days to finish the wall. Can we build the wall in a shorter period of time? Yes, if we hire more masons. Ideally, we can hire 30 masons and finish it in one day. Of course, there are practical limitations to the number of masons we can add, such as the size of the wall, the space in which they are operating together, and the way they access clay and bricks. If you put too many cooks in one kitchen, you end up having the cooks bumping to each other, arguing, moving from each other's way, without doing any actual work. This is called *thrashing*.

So the idea is simple: put more CPUs in the machine. Dual core machine have two CPUs (two cores) on the same chip. These CPUs share the same cache and of course the rest of the components as shown in the next sketch.

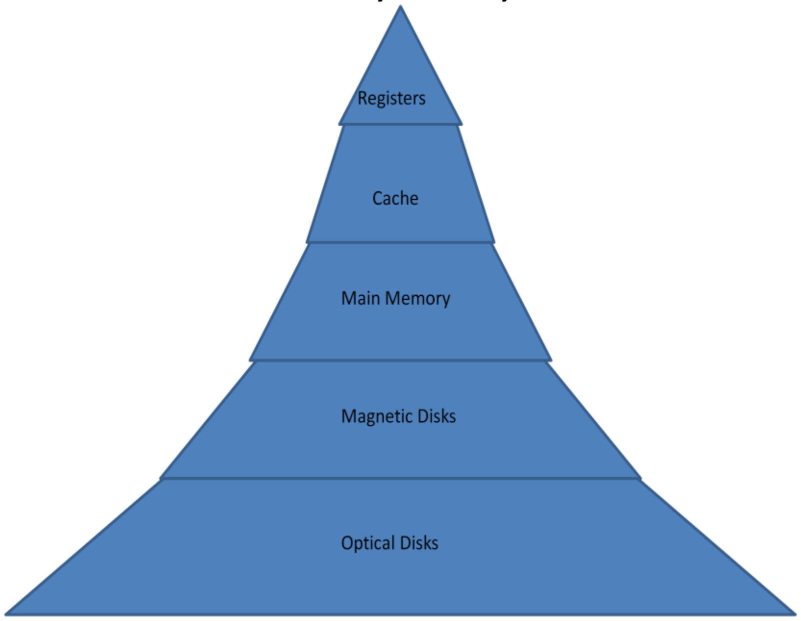
Dual-Core Architecture



Storage Units: a *bi*nary digi*t*, a *bit*, represents a 0 or a 1. A byte is 8 bits. A Kilobyte is 1024 bytes (not a 1000, with computers everything has to be a power of two, 1024 is 2^{10}). A Megabyte = 1024 Kilobytes or 1,048,567 bytes. A Gigabyte = 1024 Megabytes or 1,073,741,824 bytes. A Terabyte = 1024 Gigabyte or 1,099,511,627,776 bytes.

Memory Hierarchy: Memory can be arranged to a hierarchy.

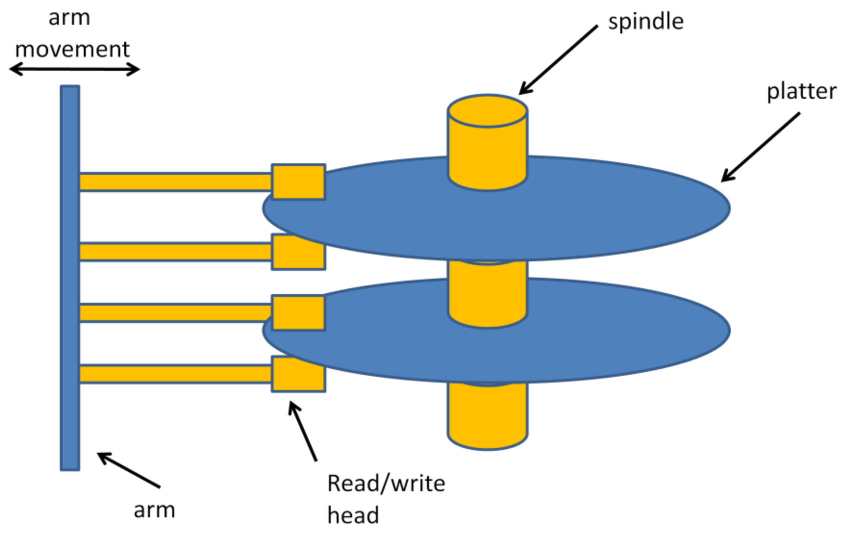
Memory Hierarchy



As we go up this hierarchy the price **per bit** becomes higher. Registers are more expensive than caches, and caches are more expensive than main memory, which is more expensive than a hard disk, and a hard disk is more expensive than a DVD. The same applies to speed. Things get slower as we move down the hierarchy. For instance, registers are very fast since they are part of the CPU itself. Cache memory is faster than main memory, but slower than registers, and henceforth. Finally, the size shrinks as we go up the hierarchy. Registers are typically several bytes in size; the cache is several Kilobytes in size; main memory is a couple of Gigabytes; a hard disk is a several tens of Gigabytes; and the pile of CDs and DVDs you have at your desk is much larger than that.

Hard disk operation: A hard disk consists of a collection of double-sided platters, typically made from some hard metal (thus the name). These platters are mounted on a spindle which rotates, rotating with it the platters.

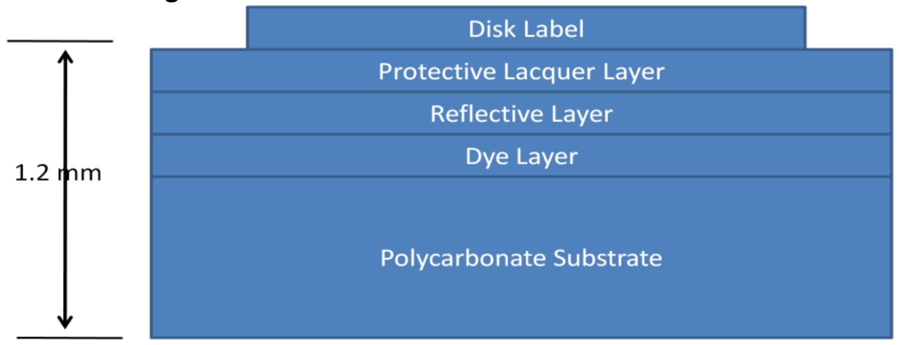
Hard disk operation



Information is stored on the surfaces of the platters as electro-magnetic signals, which are read or written by read/write heads. There is one such head for each surface. A head comes very close to the surface, without touching it, and it can sniff the electro-magnetic signals (reading) or spit them to the surface (writing). The heads are attached to one arm that moves inwards and outwards in the direction of the spindle, positioning the heads at a desired radius from the center of the platters. The spindle moves the platters so that the required location falls underneath the read/write head.

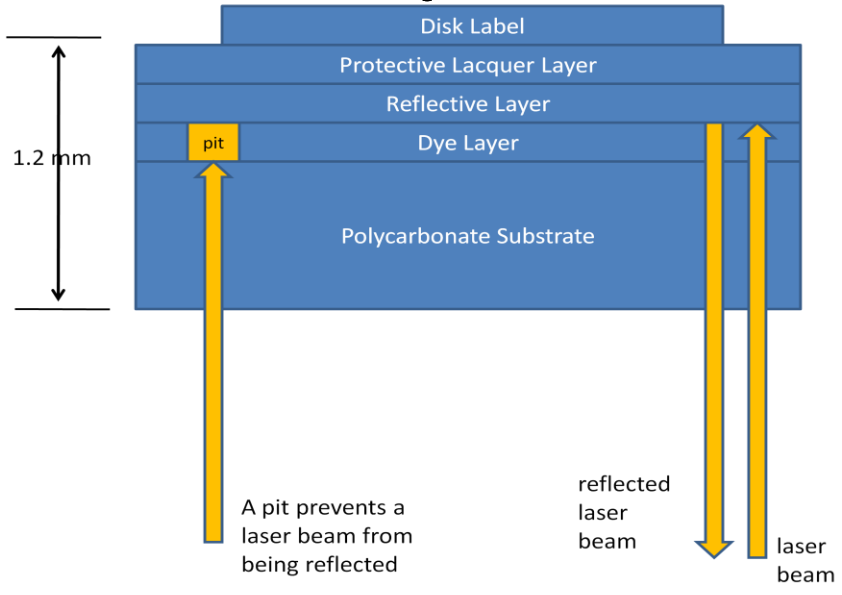
Optical disk operation: Optical disks, such as music CDs and DVDs, do not use electro-magnetic nor electric signals to store information. An optical disk has a thin dye layer of metal protected by a layer of plastic.

Magnified Cross-sectional sketch of a CD-Recordable



The bits are literally **burnt** into this dye layer. A hot laser beam burns holes into the dye layer, resulting in a spiral structure of pits (holes) and lands (no hole). Since each can represent a 0 or a 1, a colder laser beam can read this information back, without affecting the dye. If the beam escapes the dye layer and is reflected back by the reflective layer, this means a 1; if the dye is burnt, a dark spot results, preventing the beam from reaching the reflective layer and reflected back this means a 0.

Reading a CD-R



How about re-writable disks? A medium hot laser beam can be used to heat the die layer so that all pits disappear. Dual layer disks? These use two layers of different dyes so that the laser beam, if pointed at the right angle, can get to one of the two layers as if the other layer does not exist.

II – Binary Numbers

Before we have a closer look at how to build logic circuits, such as a CPU, we first need to understand the binary number system since everything inside the machine is either a 0 or a 1. In general, number systems provide a limited number of symbols that are used to represent an infinite amount of numbers.

EXAMPLE 1 – Decimal number system

The decimal number system provides only ten symbols, 0 to 9. Yet, large numbers such as 19847473737 can be represented using these 10 symbols. We start constructing numbers using one symbol only:
0
1
2
3
4
5
6
7
8
9
When we run out single-symbol numbers, we start forming numbers with two digits
10
11
12
…
When we run out of two-symbol numbers, we start forming numbers with three digits, and henceforth. Of course, you can always add 0s to the left of a number, without changing it.

The binary number system provides two symbols only: 0 and 1. The single-symbol numbers that can be constructed are simply:
0 (for zero)
1 (for one)
The next number, 2, need to be constructed from two symbols:
10 (for 2)
11 (for 3)
The next number, 4, must be constructed using three bits:
100 (for 4)
101 (for 5)
110 (for 6)
111 (for 7)
8 needs four symbols and so on.

Converting binary to decimal: If you are given a large binary number such as 10101, how do we know which decimal number it represents? Follow the algorithm:
Given a binary number of *n* bits

Number the bits in the binary number, right to left from 0 to *n-1*
The resulting decimal number is: [bit at index 0]×2^0 + [bit at index 1]×2^1 + [bit at index 2]×2^2 + [bit at index *n-1*]×2^{n-1} .

EXAMPLE 2 – Binary to decimal

The binary number 10101 is 21 in decimal. To distinguish between 10101 (21 in binary) and one thousand one hundred and one (10101 in decimal), we write the number system as a subscript. $(10101)_2$ is $(21)_{10}$.

Following the algorithm, we number the bits of 10101 right to left, as follows:
1 0 1 0 1
4 3 2 1 0 (index of bits)

The decimal number is [bit at index 0]×2^0 + [bit at index 1]×2^1 + [bit at index 2]×2^2 + [bit at index 3]×2^3+ [bit at index 4]×2^4
= [1]×2^0 + [0]×2^1 + [1]×2^2 + [0]×2^3+ [1]×2^4
= [1]×2^0 + [1]×2^2 + [1]×2^4
= [1]×2^0 + [1]×2^2 + [1]×2^4
= $2^0 + 2^2 + 2^4$
= 1 + 4 + 16
= 21

That is, $(10101)_2 = (21)_{10}$.

Note that the conversion algorithm could have been written as follows.

Given a binary number of *n* bits:
Number the bits in the binary number, right to left from 0 to *n-1*
Decimal number = 0
For each bit with a value of 1 at index i, add 2^i to the decimal number

EXAMPLE 3 – Binary to decimal

$(11010)_2$ is $(26)_{10}$.

Following the algorithm, we number the bits of 11010 right to left, as follows:
1 1 0 1 0
4 3 2 1 0 (index of bits)

The decimal number is $2^1 + 2^3 + 2^4$, since only indices 1, 3 and 4 has **1**s in the binary number
= $2^1 + 2^3 + 2^4$
= 2 + 8 + 16
= 26

Converting decimal to binary: Given a decimal number, it is converted to an equivalent binary number as follows:
Q = decimal number
Divide Q by 2, record the remainder and set Q to the quotient
Repeat step 2, until Q = 0
Binary number = the remainder bits, the most recent is the most significant

EXAMPLE 4 – Decimal to binary

What is $(18)_{10}$ in binary? Following the algorithm:
Q = 18
Dividing 18 by 2, gives **0** remainder and 9 quotient
Q = 9
Dividing 9 by 2, gives **1** remainder and 4 quotient
Q = 4
Dividing 4 by 2, gives **0** remainder and 2 quotient
Q = 2
Dividing 2 by 2, gives **0** remainder and 1 quotient
Q = 1
Dividing 1 by 2, gives **1** remainder and 0 quotient
Q = 0, stop

Binary number is **10010**

Binary addition: Addition in binary follows the same principles of addition with decimal numbers. You only have to keep in mind that we have two symbols. Basically:
0+0 = 0
0+1 = 1
1+0 = 1
1+1 = 10 (which is two in the binary number system)

EXAMPLE 5 – Binary addition

What is the result of adding 10011 and 00101

$$\begin{array}{l} 10011 \\ \underline{00101\ +} \\ \mathbf{11000} \end{array}$$

Starting at the least significant bits (from the right),

$$\begin{array}{l} \quad\ \ \mathbf{1} \\ 1\ 0\ 0\ 1\ 1 \\ \underline{0\ 0\ 1\ 0\ 1\ +} \\ \qquad\ \ \mathbf{0} \end{array}$$

1+1 is 10 or 0 and a carry of 1

$$\begin{array}{cccccc}
 & & \mathbf{1} & 1 & & \\
1 & 0 & 0 & 1 & 1 & \\
\underline{0} & \underline{0} & \underline{1} & \underline{0} & \underline{1} & \underline{+} \\
 & & & \mathbf{0} & 0 &
\end{array}$$

Then,

$$\begin{array}{cccccc}
 & \mathbf{1} & 1 & 1 & & \\
1 & 0 & 0 & 1 & 1 & \\
\underline{0} & \underline{0} & \underline{1} & \underline{0} & \underline{1} & \underline{+} \\
 & & \mathbf{0} & 0 & 0 &
\end{array}$$

Next,

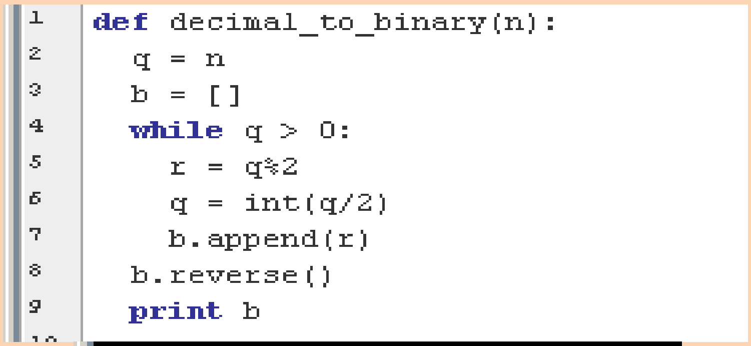
$$\begin{array}{cccccc}
 & 1 & 1 & 1 & & \\
1 & 0 & 0 & 1 & 1 & \\
\underline{0} & \underline{0} & \underline{1} & \underline{0} & \underline{1} & \underline{+} \\
 & \mathbf{1} & 0 & 0 & 0 &
\end{array}$$

Finally,

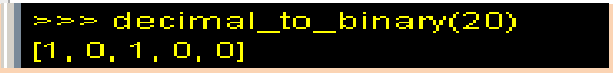
$$\begin{array}{cccccc}
 & 1 & 1 & 1 & & \\
1 & 0 & 0 & 1 & 1 & \\
\underline{0} & \underline{0} & \underline{1} & \underline{0} & \underline{1} & \underline{+} \\
\mathbf{1} & 1 & 0 & 0 & 0 &
\end{array}$$

It is left as an exercise to verify that $(10011)_2$ is $(19)_{10}$, $(101)_2$ is $(5)_{10}$, and $(11000)_2$ is $(24)_{10}$

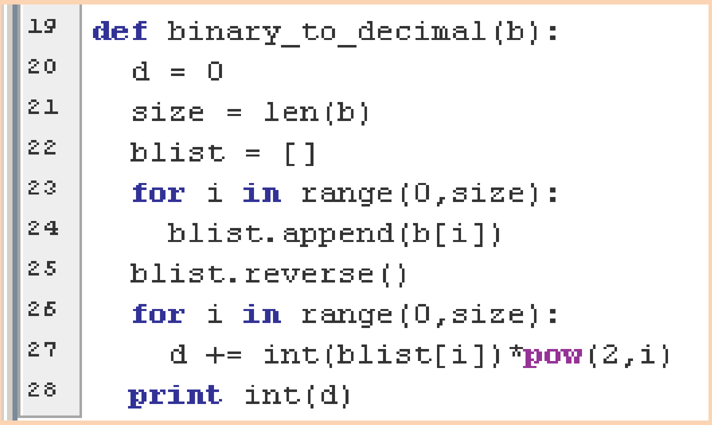
ADVANCED TOPIC – Converting between Decimal and binary in Jython



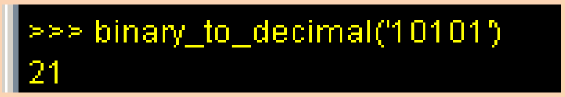
```
def decimal_to_binary(n):
  q = n
  b = []
  while q > 0:
    r = q%2
    q = int(q/2)
    b.append(r)
  b.reverse()
  print b
```



```
>>> decimal_to_binary(20)
[1, 0, 1, 0, 0]
```

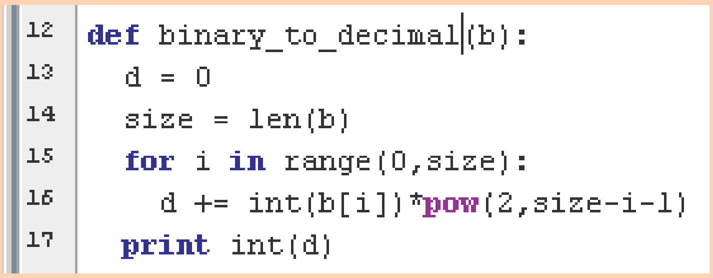


```
def binary_to_decimal(b):
  d = 0
  size = len(b)
  blist = []
  for i in range(0,size):
    blist.append(b[i])
  blist.reverse()
  for i in range(0,size):
    d += int(blist[i])*pow(2,i)
  print int(d)
```



```
>>> binary_to_decimal('10101')
21
```

Alternatively,



```
def binary_to_decimal(b):
  d = 0
  size = len(b)
  for i in range(0,size):
    d += int(b[i])*pow(2,size-i-1)
  print int(d)
```

Negative binary numbers: One famous way to represent negative numbers in binary is called the *2's complement*, and the algorithm works as follows.

Given a (positive) binary number N:
Toggle each bit in N (a 1 becomes a 0 and a 0 becomes a 1), this is called the *1's complement*
Add 1 to the 1's complement representation
Chuck out the last carry, if there is one

EXAMPLE 6 – Negative binary numbers

$(101)_2$ is $(5)_{10}$
$-(5)_{10}$ is constructed in binary as follows:
Toggle each bit of 101, the 1's complement is 010
Then add 1 to 010, resulting in 011
That is $(-5)_{10}$ is $-(011)_2$

Binary subtraction: To perform subtraction in binary, the algorithm is simple.

Given two binary numbers A and B, to calculate A – B:
Convert B into the 2's complement form
Add A and the 2's complement of B
Chuck out the carry resulting from the adding the most significant bits, if it exists

EXAMPLE 7 – Binary number subtraction

Since 5 – 5 is 0, we expect 101 + 011 to be also 0 (recall that 011 is negative 5 in 2's complement)

$$\begin{array}{l} 1\,0\,1 \\ \underline{0\,1\,1\,+} \\ 0\,0\,0 \end{array}$$

(Note that the last carry resulting from adding the most significant bits is chucked out)

Binary multiplication: As we mentioned earlier, multiplication in any number system can be done using addition. The algorithm is given next.

Given two (binary) numbers A and B, the multiplication of A and B is calculated as follows:
If A < B, then add B to itself A times
Else add A to itself B times

The if statement tries to minimize the number of iterations in the algorithm. However, in general, it does not really matter whether we add A to itself B times or B to itself A times, the result is always the same.

EXAMPLE 8 – multiplication by addition

To calculate 2×3, we can either:
Add 2 to itself 3 times, or
Add 3 to itself 2 times

The first option gives us: 2+2+2 = 6
The second: 3+3 = 6

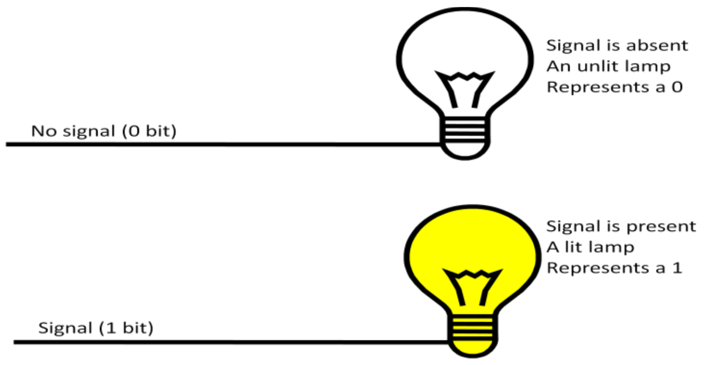
Our algorithm chooses option 2, since this will take less time to execute by the computer (one fewer step). In general when the numbers are much larger, savings would be much larger. For instance, to multiply 2 and 100, the first option would require us to add 2 to itself 100 times; this is 100 addition operations. Option 2, only requires two addition operations, adding 100 to 100.

III –Boolean Logic Circuits

Now we are ready to have finer look at how computer hardware is built. Computer hardware can be built using any number system. The reason the binary system is always used is that it makes the design simpler and simpler designs are less prone to error. Furthermore, building a computer using the binary number system can be done electronically, mechanically (using levers and gears), or even thermodynamically (using fluids and pistons). The choice of electronics to build computers results in a much smaller machine.

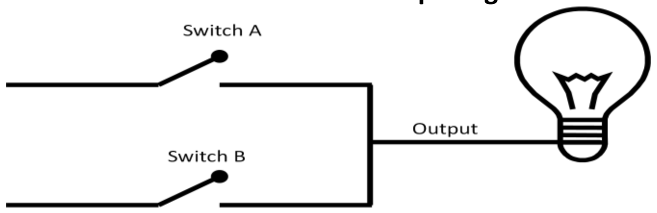
Representing signals: A signal in a single wire can represent a 1, so that the absence of a signal is interpreted as a 0. Computers use two levels of voltages to represent 1s and 0s. A small voltage corresponds to a 0 and a high voltage corresponds to a 1. It is always more convenient to think about it as present or absence signal.

Abstraction of presence/absence of signals as 1s and 0s



Parallel and serial signals: The basic building blocks of a computer are based on the logic operations: **and**, **or**, and **not**. Everything else can be built from these as we shall see later. A circuit with output signal generated from two input signals can be built in one of two ways: parallel or serial.

Parallel combination of input signals



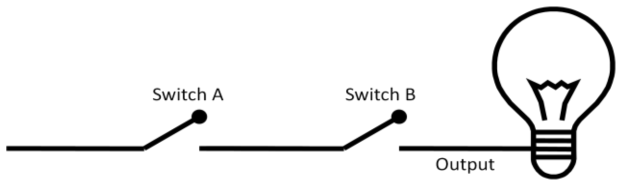
This circuit can light the lamp if any of the switches A or B is closed. This will allow the input current to flow to the output, causing the lamp to go on. However, if both switches are open, the lamp remains off. It should be obvious that this can be abstracted by an **or** function. If each input switch is represented by a 0 (for open) and 1 (for closed), the output is 1(lit lamp) when at least one of the inputs is 1; the output is 0 (off lamp) as long as both switches are open (0). We reproduce the **or** truth table here, replacing T with 1 and F with 0.

Truth table for or

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

The serial combination of input signals is abstracted by the **and** function. Here, both switches A and B must be closed (1) for the output to be 1 and the lamp to light. Otherwise, any open switch will deny the current from passing to the lamp.

Serial combination of input signals



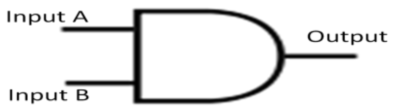
The **and** truth table is reproduced below, replacing 1 for T and 0 for F.

Truth table for and

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

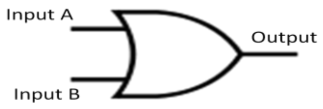
The **not** function abstracts an electrical that is called an inverter. It will simply cause the switch to operate backwards: the current flows if the switch is open and it does not if the switch is closed. Explaining the details of inverters is beyond the scope of our discussion.

Logic gates: Luckily, we do not need to think about invertors, parallel, and serial circuits. These can be simply abstracted using the Boolean logic operations, **and**, **or** , and **not**. A serial circuit is represented using an *and gate* with the symbol:



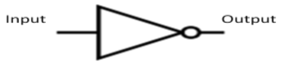
The output is 1 if and only if both inputs A and B are 1s; otherwise, the output is 0.

A parallel circuit is represented using an *or gate* with the symbol:



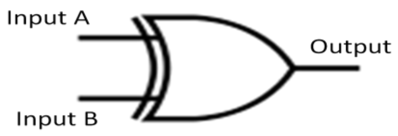
The output of an or gate is 0 if and only if inputs A and B are 0s; otherwise, the output is 1.

Finally an inverter circuit is represented using a *not gate*:



The output is 1 only if the input is 0 and vice versa.

While these three are sufficient to build any logic circuit, including a full blown ALU, it is sometimes beneficial to add more gates so that the design is simplified. One such gate is the *xor gate*:



The xor's gate output is 1 as long as exactly one of the inputs (A or B) is 1; otherwise it is a 0. The **xor** truth table is reproduced below, replacing T with 1 and F with 0.

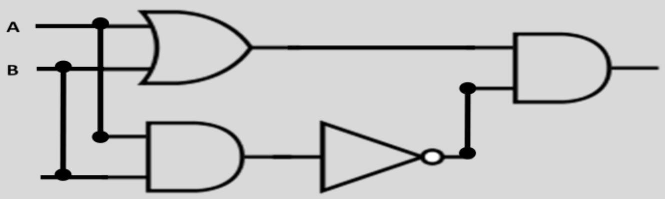
Truth table for xor

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

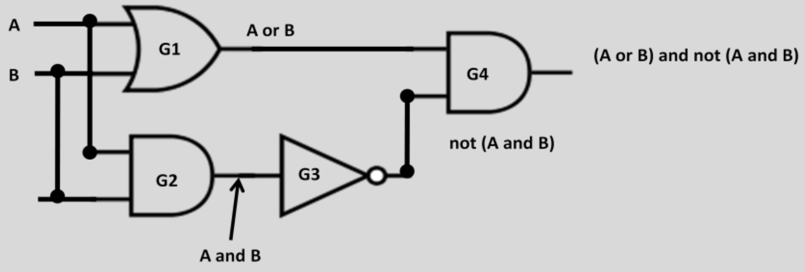
Logic circuits: Any logic circuit can be built from these basic building blocks. For instance, the xor gate can be built as a circuit involving **and**, **or**, and **not** gates.

EXAMPLE 8 – Building a xor from other gates

Recall that A XOR B is logically equivalent to (A OR B) AND NOT (A AND B). This can be represented by the following circuit:



The operation of this circuit is illustrated below:



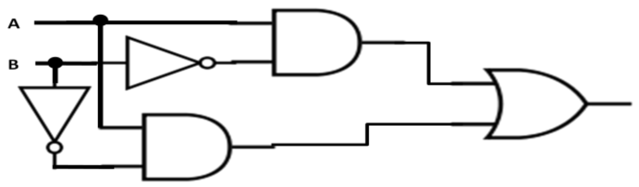
Gate G1 is an OR gate, so its output is the **or** of its inputs (A OR B). G2 is an AND gate and its inputs are pinned on A and B; hence, its output is (A AND B). G3 inverts or negates the output of G2, producing NOT (A AND B). Finally the outputs of G1 and G3 are fed as inputs to G4, which is an AND gate, resulting in (A OR B) AND NOT (A AND B).

Circuits from truth tables: An easy way to construct a logic circuit is to first write down the outputs for all possible inputs. This can be done using a truth table. Take for instance the XOR truth table discussed earlier. Examining the table, we only have two possibilities where the output is 1, namely:

A	B	A XOR B
0	1	1
1	0	1

We then write a logic formula for each row, consisting of AND and NOT operations, such that the formula is true for the given input combination. For the first row, where A = 0 and B = 1, the formula is (NOT A) AND B. For the second row, the formula is A AND (NOT B). Then we take the OR of all such formulas:
((NOT A) AND B) OR (A AND (NOT B))

This formula is expressed using the following circuit:



That is this is an alternative way to represent a XOR circuit. The general algorithm for converting a truth table to a logic circuit is as follows.

Given a truth table:

1. For each output column in the truth table, for each row R in the truth table where the output is 1, Start with an empty formula F_R for row R:
    a. For each input value X in R that is 1, F_R = F_R AND X
    b. For each input value X in R that is 0, F_R = F_R AND (NOT X)
2. OR all the formulas F_R for all rows R (the result is called the *Boolean sum of products*)

EXAMPLE 9 – Converting a truth table to a logic formula

Given the following truth table:

A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Step 1 of the algorithm is to identify all rows with an output of 1. These are

A	B	C	Output
0	1	0	1
0	1	1	1
1	0	1	1
1	1	1	1

The formula for row 1 is (NOT A) AND B and (NOT C)
For row 2: (NOT A) AND B AND C
For row 3: A AND (NOT B) AND C
For row 4: A AND B AND C

The Boolean sum of products is : [(NOT A) AND B AND (NOT C)] OR [(NOT A) AND B AND C] OR [A AND (NOT B) AND C] OR [A AND B AND C]. The circuit now can be drawn.

Simplification: The circuit of Example 9 would be complex to draw. Fortunately, logic formulas can be simplified in a manner similar to arithmetic formulas. Take the expression generated in Example 9:

[(NOT A) AND B AND (NOT C)] OR [(NOT A) AND B AND C] OR [A AND (NOT B) AND C] OR [A AND B AND C]
For simplicity we can re-write it as follows:
$A^{-1}BC^{-1} + A^{-1}BC + AB^{-1}C + ABC$, where the + is the OR, X^{-1} is NOT X, and XY is X AND Y
Now we can factor it as if it is an arithmetic expression:
$A^{-1}BC^{-1} + A^{-1}BC + AB^{-1}C + ABC$
$= A^{-1}B(C^{-1} + C) + AC(B^{-1} + B)$
$= A^{-1}B(1) + AC(1)$, since $X^{-1} + X$ is a tautology (keep in mind that this is NOT X OR X)
$= A^{-1}B + AC$
This formula is equivalent to the original complex formula, but the resulting circuit can be much simpler. We will draw the logic circuit in Example 12.

Adders: Adding two bits can be also done using these basic gates. Let's examine all possible combinations for adding two bits, using the following truth table:

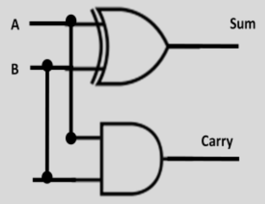
Truth table adding two bits

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

We can use the Boolean sum of products for the two output columns: Sum and Carry. For the Carry column the Boolean sum of products us simply A AND B. For the Sum, it is ((NOT A) AND B) OR (B AND (NOT A)), which can be expressed as A XOR B.

EXAMPLE 10 – one-bit half adder

A one bit adder can add to numbers of size one bit each. The following circuit can add two bits:



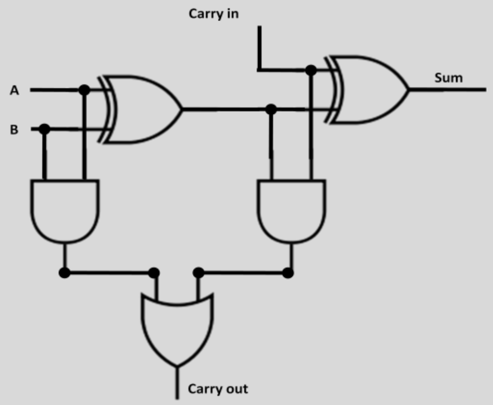
The adder of Example 10 is insufficient if we would like to add numbers of sizes larger than one bit.

EXAMPLE 11 – one-bit full adder

Assume we are trying to add the numbers 11 and 11 as follows:
11
<u>11+</u>

Adding the first two bits (the least significant ones) can be done using a half adder, resulting in 0 for the sum and 1 for the carry. Now we have the following situation:
1
11
<u>11+</u>
 0

To add the remaining two bits, a half adder is insufficient because we need to take into account the carry out from the previous addition operation. A half adder has only two inputs, there is no third input for the carry in. A full adder fixes this problem.

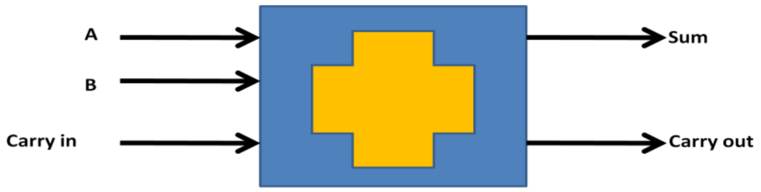


The input to this circuit is three bits: A, B and the carry in. The output is the Sum of A and B and the carry out. Notice that if the carry in is always set to 0, the circuit becomes equivalent to a half adder.

The truth table corresponding to the full adder has 8 different combinations (since we have three inputs and each input can have two values, the combinations are 2^3). The truth table for the full adder is left as an exercise. You should also attempt constructing the full adder circuit using the Boolean sum of products as an exercise.

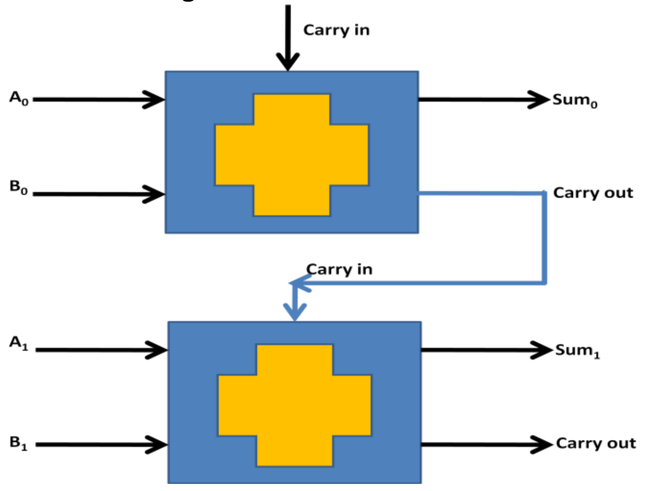
So how do we construct an n-bit adder so that it adds n-bit numbers. Through the power of abstraction, this is straightforward. We can treat the full adder as a black-box, it accepts three inputs: A, B, and the carry in and produces two outputs: Sum = A+B and the carry out. We understand how this is done, but we do not care about the details at this stage. We are hiding them intentionally.

Abstraction of a 1-bit full adder



To construct a two-bit adder, we need to connect two 1-bit adders so that the carry out of the adder corresponding to the least significant bit is connected to the carry in of the second adder.

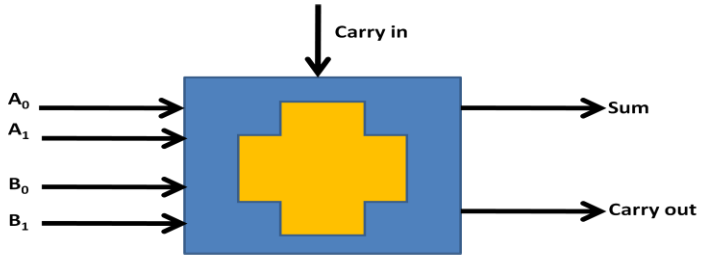
Constructing a 2-bit adder from two 1-bit adders



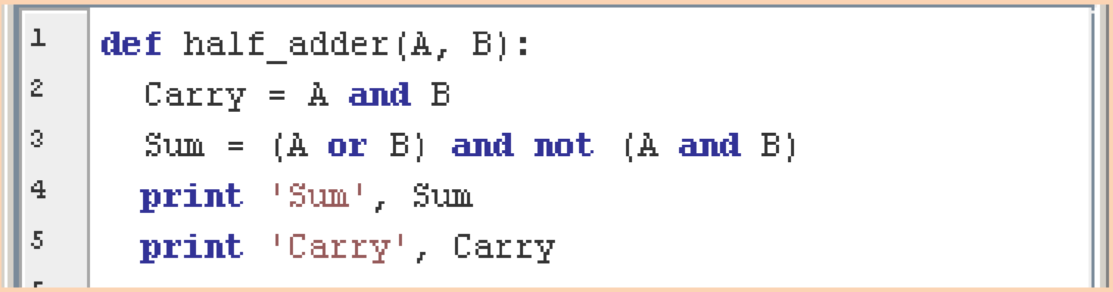
Two numbers A_1A_0 and B_1B_0 can now be added. A0 and B0 are added using the top adder, and if there is a carry it is fed to the next adder which adds A_1 and B_1. The result of the addition is Sum_1 Sum_0 with carry out. This approach can be generalized to build an n-bit adder, for any n.

Notice that in the two-bit adder, the carry out of the top 1-bit adder is no longer an output of the 2-bit full adder. The 2-bit full adder can be abstracted as the following black box.

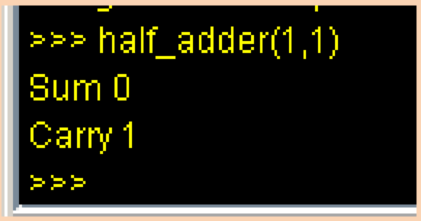
Abstraction of a two-bit full adder



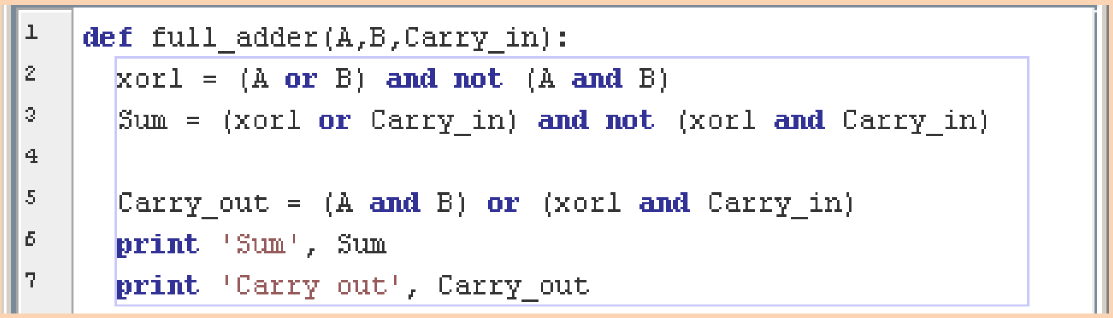
ADVANCED TOPIC – 1-bit adders in Jython



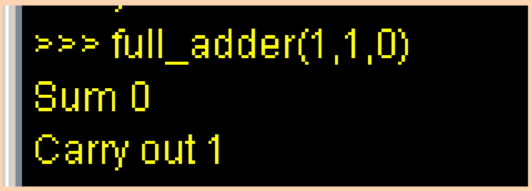
```
def half_adder(A, B):
  Carry = A and B
  Sum = (A or B) and not (A and B)
  print 'Sum', Sum
  print 'Carry', Carry
```



```
>>> half_adder(1,1)
Sum 0
Carry 1
>>>
```



```
def full_adder(A,B,Carry_in):
  xor1 = (A or B) and not (A and B)
  Sum = (xor1 or Carry_in) and not (xor1 and Carry_in)

  Carry_out = (A and B) or (xor1 and Carry_in)
  print 'Sum', Sum
  print 'Carry out', Carry_out
```

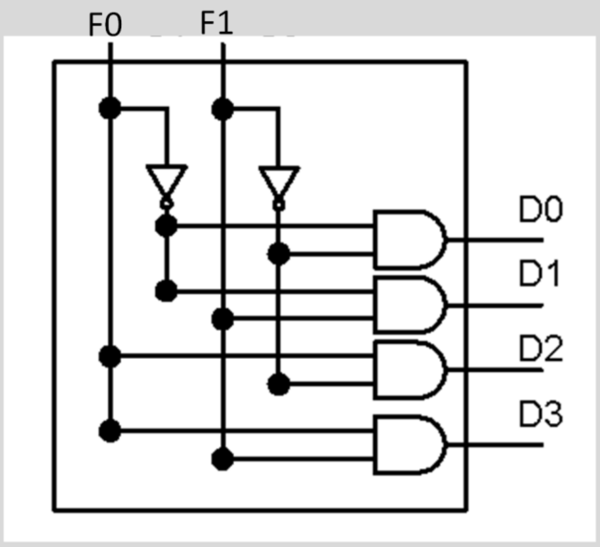


```
>>> full_adder(1,1,0)
Sum 0
Carry out 1
```

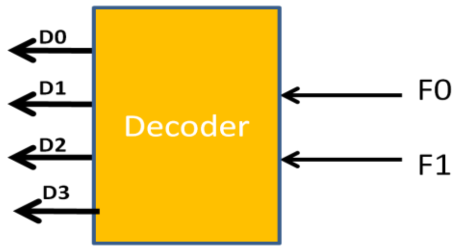
Decoders: A decoder is a logic circuit that has n inputs and 2^n outputs such that in every possible combination of inputs a unique output is set to 1 and the rest to 0.

EXAMPLE 12 – 2-4 decoder

This decoder has two inputs F1 and F0, and produces outputs D0 – D3
When both inputs are 0s, the output D0 is 1 and the rest (D1 to D3) are 0s.
When F0 is 0 and F1 is 1, only D1 is 1, and the rest are 0s.
When F0 is 1 and F 1 is 0, only D2 is 1, and the rest are 0s.
When both inputs are 1s, only D3 is 1, and the rest are 0s.



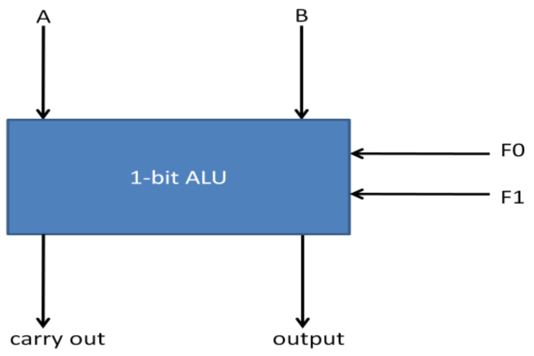
This decoder can be abstracted as a black box:



ALU logic circuit: Now we have all the required components to build a one-bit ALU, an ALU that operate on inputs of 1-bit length. The same technique that was used to build a general adder from the 1-bit full adders can be used to build an n-bit ALU, for any n.

Our example ALU performs four operations only: **and**, **or**, **not**, and **addition**. So, we need a 2-4 decoder, such as the one of Example 12, to instruct the ALU which of the four operations is required. We will use a top-down design approach here, starting at the ALU block diagram (black box) and then refining it by looking into its details.

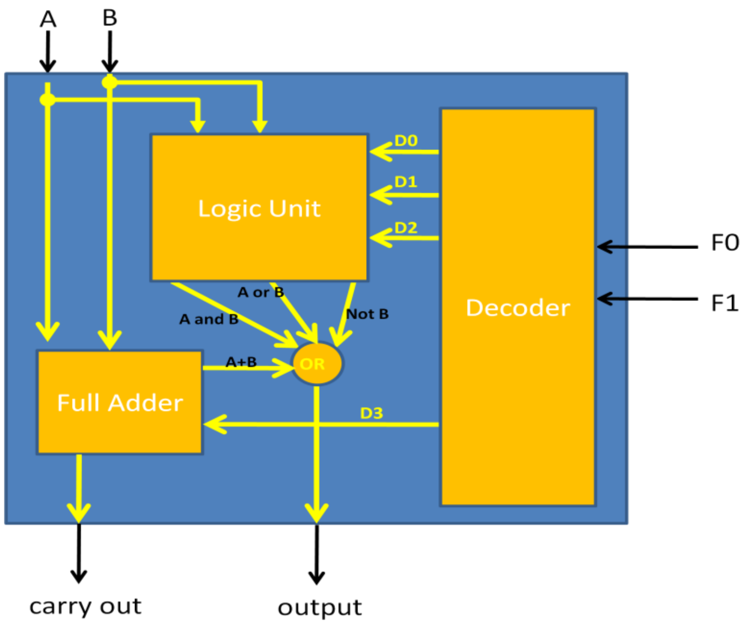
High-level design of a 1-bit ALU



Our ALU receives four inputs (bits): A and B are the input bits, loaded from the ALU input registers. The function inputs F0 and F1 are received from the CU and determine the type of operation and output required from the ALU.
A function input of 00 (F0 = 0 and F1 = 0) produces output = A and B.
A function input of 01 (F0 = 0 and F1 = 1) produces output = A or B.
A function input of 10 (F0 = 1 and F1 = 0) produces output = not B.
A function input of 11 (F0 = 1 and F1 = 1) produces output = A + B, with the carry value in the output carry out.

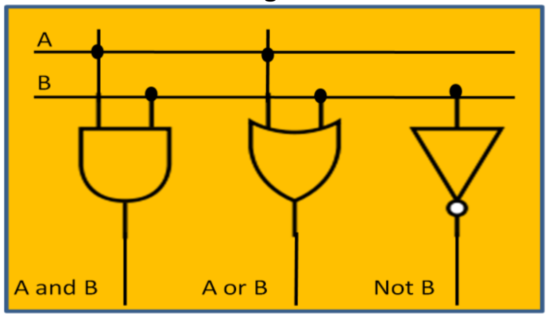
The three components of our ALU are: a full adder to perform addition, a logic unit with the three basic logic operations (**and**, **or**, and **not**), and a 2-4 decoder that controls the output of the ALU.

Refining the 1-bit ALU



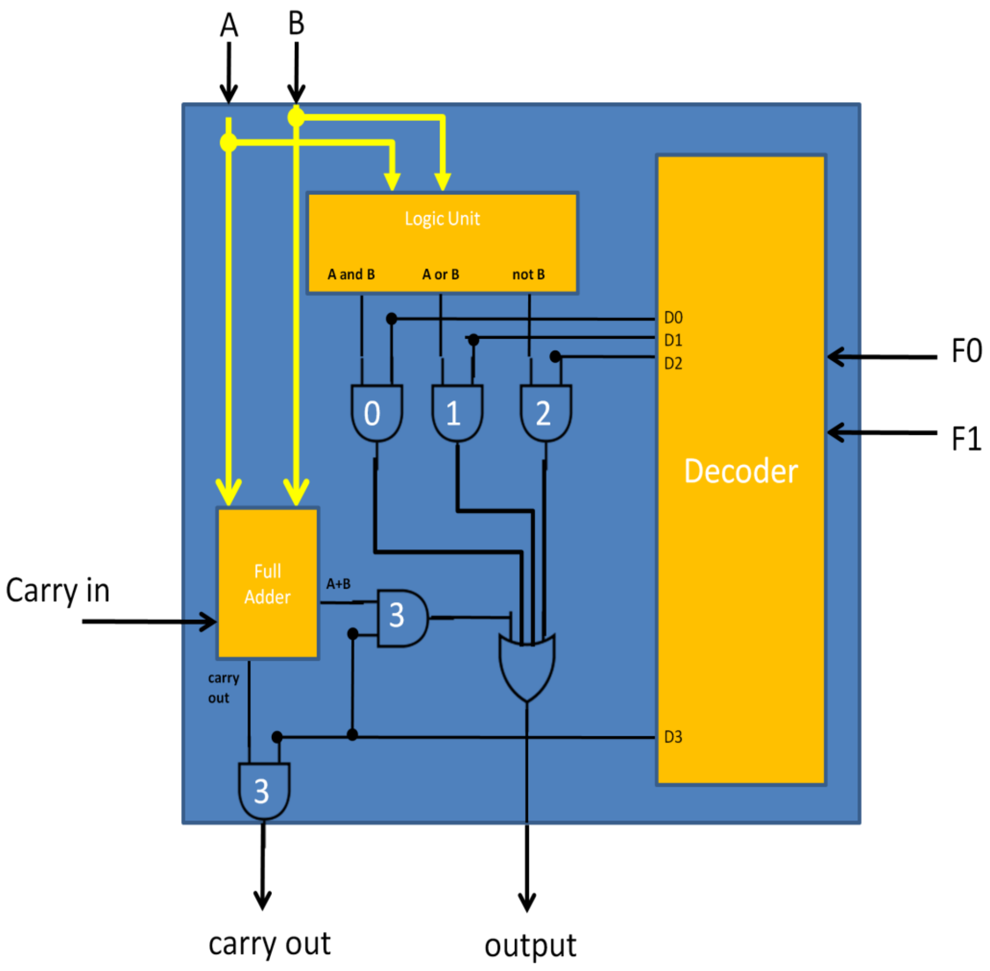
Three of the decoder outputs (D0-D2) are wired to the logic unit to determine if the operation is one of: **and**, **or**, or **not**. The fourth (D3) is wired to the full adder. We have already built the decoder and the full adder. The logic unit is also straightforward.

The logic unit



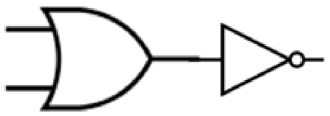
The trick is connecting the decoder to the rest of the components. This is also simple. We will “and” each of the decoder outputs to the result of each of the four ALU functions. For instance, if we “and” D3 with the “sum” output of the full adder, the sum will be output only if D3 is 1 (the sum function is selected). Recall that only one of the outputs of a decoder will be 1 and the rest are zeroed.

Connecting the decoder to the rest of the ALU components

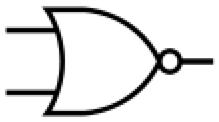


If the decoder enables D0 (D0 = 1), then D1-D3 are all disabled (0s). With the exception of the **and** gate labeled with 0, the output of all of the **and** gates will be 0. Hence, the output of the **or** gate is solely determined by the result “A and B”. That is the ALU’s output is “A and B”.
If the decoder enables D1 (D1 = 1), then D0, D2, and D3 are all disabled (0s). With the exception of the **and** gate labeled with 1, the output of all of the **and** gates will be 0. Hence, the output of the **or** gate is solely determined by the result “A or B”. That is the ALU’s output is “A or B”.
If the decoder enables D2 (D2 = 1), then D0, D1, and D3 are all disabled (0s). With the exception of the **and** gate labeled with 2, the output of all of the **and** gates will be 0. Hence, the output of the **or** gate is solely determined by the result “not B”. That is the ALU’s output is “not B”.
If the decoder enables D3 (D3 = 1), then D0-D1 are all disabled (0s). With the exception of the two **and** gates labeled with 3, the output of all of the **and** gates, will be 0. Hence, the output of the **or** gate is solely determined by the result “A + B”. That is the ALU’s output is “A + B” and the carry out is also allowed out.

Memory circuits: It remains to show how to use logic circuits to build a register. Main memory can be built as a collection of registers, though different technologies are employed nowadays that are not limited to registers. First we introduce NOR gates, which is simply a NOT OR gate. An OR gate and a NOT gate can be combined in the following manner:



That is, the result is NOT(A OR B), if the inputs are A and B. This can be represented using a single gate, the NOR gate:



The truth table for a NOR gate is as follows:

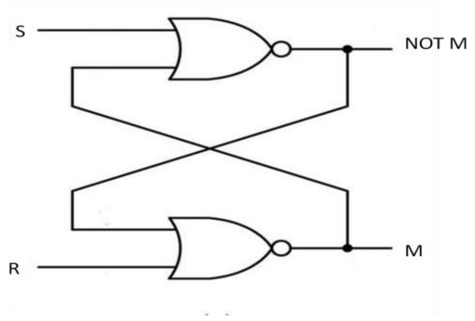
Truth table adding two bits

A	B	A OR B	NOT(A OR B)
0	0	0	**1**
0	1	1	**0**
1	0	1	**0**
1	1	1	**0**

Note that using DeMorgan’s rule, a NOR gate is equivalent to (NOT A) AND (NOT B)

Latches: The basic building block for a memory circuits is called a *latch*. This is a circuit that can remember one bit. There are different kinds of latches. The one we discuss here is called an SR (Set-Reset) latch, and it is built using two NOR gates.

SR latch



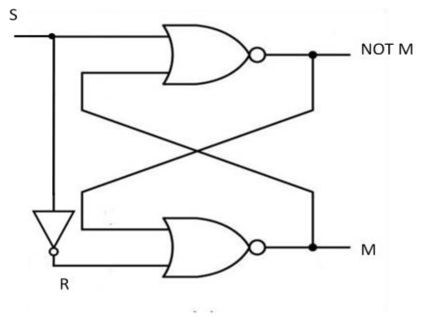
There are two inputs to the SR latch: S and R, and two outputs M and NOT M. M is the memory bit memorized by this circuit. The S input sets the output M to 1, and the R input resets the output M to 0. The SR latch circuit is tricky. Output from the top NOR gate is fed back as an input to the bottom NOR gate and vice versa.

To set the latch, S is set to 1 and R to 0. The output of the top gate (NOT M) will be 0 regardless of what the fed back input to the gate is (1 OR anything is 1, and the negation of 1 is 0). After some time, the fed back input to the bottom gate stabilizes at 0. Since R is 0 the output of the bottom gate will be 1, setting the memory bit M to 1.

To reset the latch, S is set to 0 and R to 1. The output of the bottom gate (M) will be 0 regardless of what the fed back input to the gate is (1 OR anything is 1, and the negation of 1 is 0). This resets the memory bit M to 0. After some time, the fed back input to the top gate stabilizes at 0. Since S is 0 the output of the top gate will be 1.

When both S and R have the same value, the output cannot be easily determined. To prevent errors resulting in an input where both S and R have the same value (such as having both at 1: reset and set the circuit at the same time), we can update the latch as follows:

Updated latch (S latch) with a single input

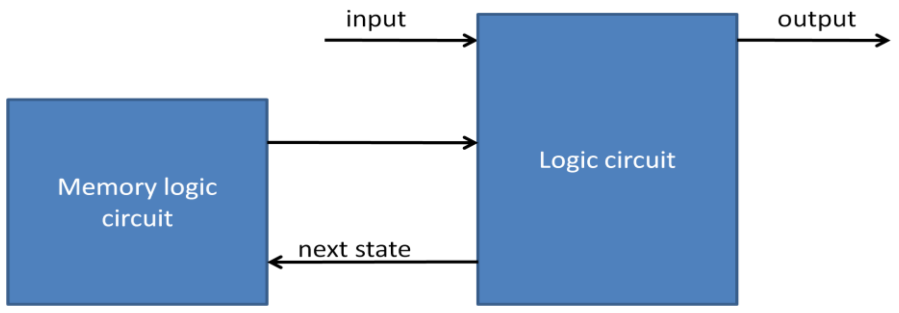


To set this latch, we simply set S to 1, making R 0. To reset it, we set S to 0, making R 1.

Multi-bit registers: Two build an n-bit register, we need to group n latches together.

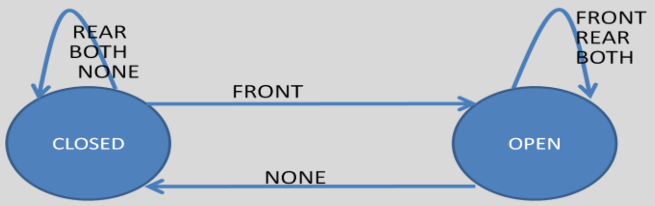
IV – Finite State Machines as Logic Circuits

FSMs change state, depending on the current state they are in and the input they receive. Hence, an FSM needs memory to remember the state it is in. An FSM can be built using Boolean logic, with a collection of registers needed to remember states.



EXAMPLE 13 – Automatic door controller

Recall the sliding door FSM from Chapter 3. Let's imagine that the door only opens to the front. The resulting FSM is given by the following diagram:



The state table is as follows:

Input	Current State	Next State
NONE	CLOSED	CLOSED
NONE	OPEN	CLOSED
FRONT	CLOSED	OPEN
FRONT	OPEN	OPEN
REAR	CLOSED	CLOSED
REAR	OPEN	OPEN
BOTH	CLOSED	CLOSED
BOTH	OPEN	OPEN

One 1-bit register is sufficient to remember the current state: 0 is closed and 1 is open.

The inputs received and their binary encodings are:

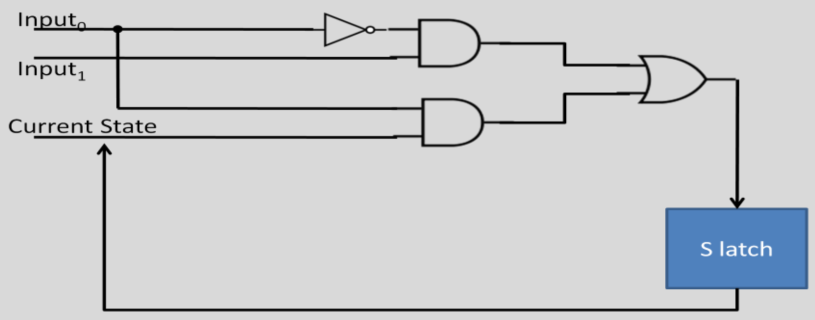
NONE	00
FRONT	01

REAR	10
BOTH	11

The state table in binary is:

Input		Current State	Next State
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

The logic formula equivalent to this truth table has been generated in Example 9. The circuit is (after applying simplification which was performed immediately after Example 9):



V – Relationship between HL and LL programs

Programmers write high-level code, such as Jython programs and computers only understand 0s and 1s. The machine (low-level) language drives the logic circuits of the computer. Compilers or interpreters translate high-level code to low level code, which is executed by the computer hardware (the logic circuits). Consider the following Jython statement, which adds to variables and stores the result in a third variable:

```
x = y + z
```

The low level code generated from this statement is nothing but a stream of 0s and 1s. Yet, we an abstract it using English-like statements and variable names (called *assembly language*). The translation of x = y + z may look like the following assembly language program:

```
LOAD y A
LOAD z B
ADD A B C
STORE C x
```

Which instructs the CPU to load memory location y to register A; load memory location z to register B; instruct the ALU to perform an addition operation storing the result in register C; finally the register C is stored back to memory location x.

At the machine-language level, each instruction (LOAD, ADD, STORE, etc…) has a fixed binary representation, just like ASCII codes for characters. The registers are referred to by their numbers and memory locations are replaced by the actual memory address, all in binary. The above program can look like the following machine language program:

```
00101001  10101010   0000001
00101001  10111000   0000010
01101011  00000001   0000010  00000011
00101010  00000011  00101010
```

LOAD is represented by 00101001; y is the memory address 10101010; A is register 1 or 0000001; z is memory address 10111000; register B is number 2 or 0000010; ADD is 01101011; register C is number 3 or 00000011; STORE is 00101010; and memory location x is 00101010. We should note that these codes are for an imaginary machine and codes can differ from one machine to another.

Fetch-decode-execute cycle: once the high-level program is translated to 0s and 1s. It is loaded to main memory from the hard disk. A special register in the CPU, called *program counter* (PC), is loaded with address of the first instruction in the program. (the name program counter is confusing since it has nothing to do with counting). The CPU fetches the first instruction from main memory (or the cache) and brings the instruction to a special register, called the *instruction register* (IR). The CU (control unit) decodes the instruction, determining the type of work needs to be done. If it is a LOAD, it gets the address of the variable to be loaded and sends an instruction to main memory to get the contents of the required location. If it is an ADD, it sets up the proper inputs to the ALU circuit so that it can perform an

ADD. Then, the instruction is carried out or executed. All of this is done in the electronic circuits that are built using logic operations.

Exercises

1. Convert each of the following binary numbers to decimal:
   101010, 100100, 111111, and 01000011
2. Convert each of the following decimal numbers to binary:
   31, 100, 1234 and 1778
3. Write the following negative decimal numbers in binary, using the 2's complement format:
   -6, -12, -21, and -78
4. Add the binary numbers: 101010 + 111111 and 00100111+11100010
5. Draw a circuit corresponding to the following logic formulas:
   a. (A AND B) OR (A AND NOT B)
   b. (A XOR C) AND (A OR (B OR NOT C))
   c. (B NOR C) AND (A XOR NOT (B OR C))
6. Convert the following truth table to a logic formula:

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

7. Simplify the formula from Exercise 6 and draw the circuit
8. Construct a truth table for a 1-bit full adder and follow the Boolean sum of products to design the circuit from scratch
9. Design a 1-2 decoder
10. Design a 3-8 decoder
11. Design a logic circuit that represents the FSM of the automatic door that opens both ways in Chapter 3.
12. What are the likely assembly language instructions corresponding to the following Jython statement: x = x + (y/z)?

Chapter 5
Taming Data Dinosaurs: Databases

Computers run programs that operate on data. When such data becomes large, finding efficient ways to organize it becomes crucial. We all organize our data, from shopping lists to monthly budget analysis. Nevertheless, large amounts of data need to be carefully designed and organized. The simple list structure of “to-do” lists falls short for larger data sets. Banks keep information about all their customers, accounts, and even all the transactions performed on each account. This is a lot of data that need to be organized so that the computer programs work properly and the bank functions.

A *database* is an organized collection of data. Your mobile phone uses a database to store your address book. The university uses a database to keep track of students, their course and grades, and much more. In this chapter, we will learn how to design data models. Such models, if properly designed, lead to properly designed databases. Once we learn how to model data and design databases, we will look at a programming language that is meant to interact with databases, called SQL.

I – Data Modeling

The data modeling approach we discuss here is called the *Entity-Relationship* model, abbreviated ER. Database can be modeled as entities and relationships between these entities.

Entities and Entity Types: An *entity* is any object that exists in the real world. Such an object can exist physically, in a tangible way, or conceptually, in a virtual way. A book, car, student, and jacket are all examples of physical entities. A course, bus route, and job are entities of conceptual existence. An *entity type* specifies a class of entities. The employee “Scott Smith” is an entity of type **employee**; so is “Sarah Smith”, say. These entities are of the same type, employee. All employees share common characteristics. An entity type defines these common characteristics. For instance all employees have names, this is a common characteristic. Two different employees may have different names, but they have names. Another common characteristic is the salary; all employees earn a salary. The value of the salary could be different between employees.

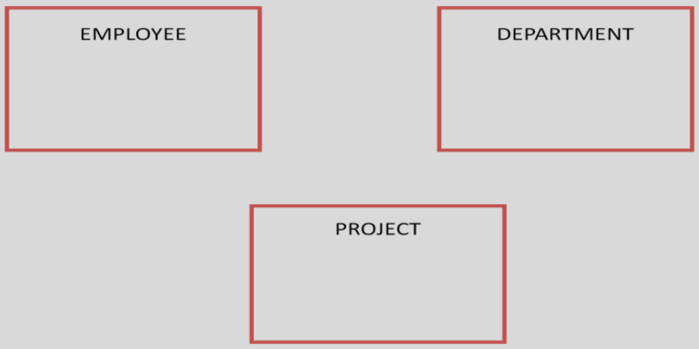
We will use as a running example a simple company, called the 203 company. It has employees working on projects and departments that manage projects.

EXAMPLE 1 – Entity Types

The entity types in the 203 company are: EMPLOYEE, PROJECT, and DEPARTMENT.

ER Diagrams and Entity Types: Our ER model is specie as a special kind of graph, called *ER diagram* (ERD). The vertices in this graph are entity types. These vertices are drawn as boxes.

EXAMPLE 2 – 203 Company ERD

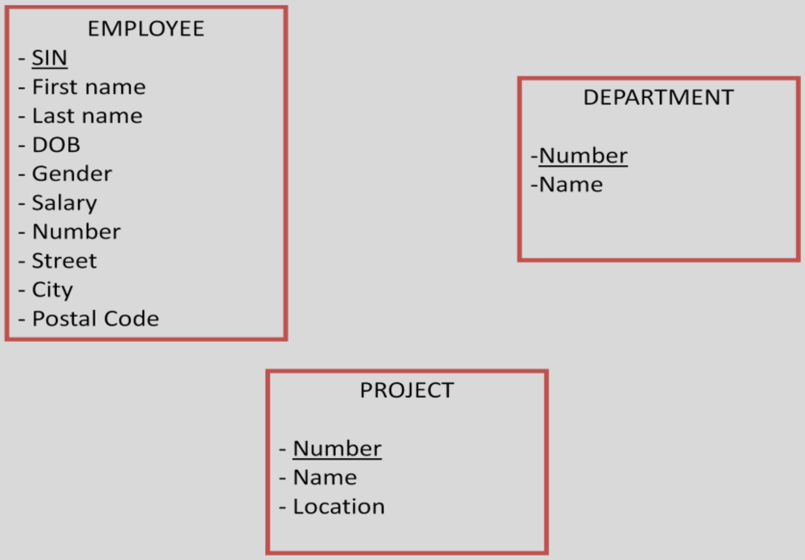


Attributes: Entities have attributes, properties that describe them. An employee for instance can be described by a name, date of birth, salary, address, etc … The choice of the attributes is determined by the problem we are trying to solve. In the ERD, the attributes of each entity type is listed inside the box corresponding to that entity type. These attributes will be assigned values, which could be different from one entity to another. A collection of attributes that uniquely identify an entity is called a *primary key*.

EXAMPLE 3 – 203 Company ERD

An employee can be described by a social insurance number (SIN), name (first and last), date of birth (DOB), gender, salary, and address (number, street, city, and postal code).

A department has a number and a name. In addition, a project has a location.



The attribute SIN of EMPLOYEE is underlined to indicate that it is a primary key for EMPLOYEE. No two different employees can have the same SIN. Similarly, Number in PROJECT is the primary key. The department Number is the primary key for DEPARTMENT.

Take a specific entity of EMPLOYEE, "Debra Beacons". It may have the following values assigned to the EMPLOYEE attributes: (171717171, Debra, Beacons, 15-Aug-1961, Female, $70,000, 15, Bacon Hill, Ham Land, T2X Y0Y). Similarly, employee "Sam Field" can be described by: (181817178, Sam, Field, 17-Feb-1978, Male, $40,000, 15, Kick Way, Ball Land, Y2K K0K)

This ERD is not complete yet. It does not contain relationship information (the R of ER). How does an employee relate to a department? How do employees relate to projects?

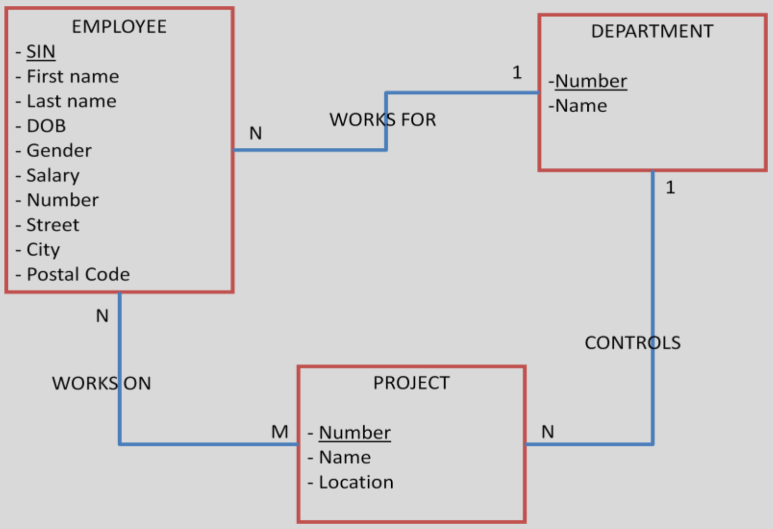
Relationships and Relationship Types: A relationship type specifies if and how entity types relate to each other. A relationship is an instance of the relationship type relating entities together. Relationship types are the edges of the ERD graph. These edges are labeled to convey extra information.

Cardinality of Relationship Types: One piece of extra information needed for the edges is the cardinality of the relationship types. We have three cardinalities:

- One-to-one: represents a unique association between entity-types. Attributes are normally regarded as one-to-one relationship types.
- One-to-many (many-to-one): An entity can be associated with more than one other entity. A department can have many employees, but an employee is allowed to belong to one department only.
- Many-to-many: An employee can work on more than one project, and a project can have many employees working on it. This is an example of a many-to many relationship type.

EXAMPLE 4 – 203 Company ERD

The ERD of the previous example is revised to include three relationship types: WORKS FOR, CONTROLS, and WORKS ON



The WORKS FOR edge is labeled with 1 on the EMPLOYEE side and N on the DEPARTMENT side. This is an indication that WORKS FOR is a one-to-many relationship type: An employee can work for one department, but a department may have many employees working for it.

The CONTROLS relationship type is also one-to-many : A department may control several projects, but a project must be controlled by a single department (inter-departmental projects are not allowed in the 203 company).

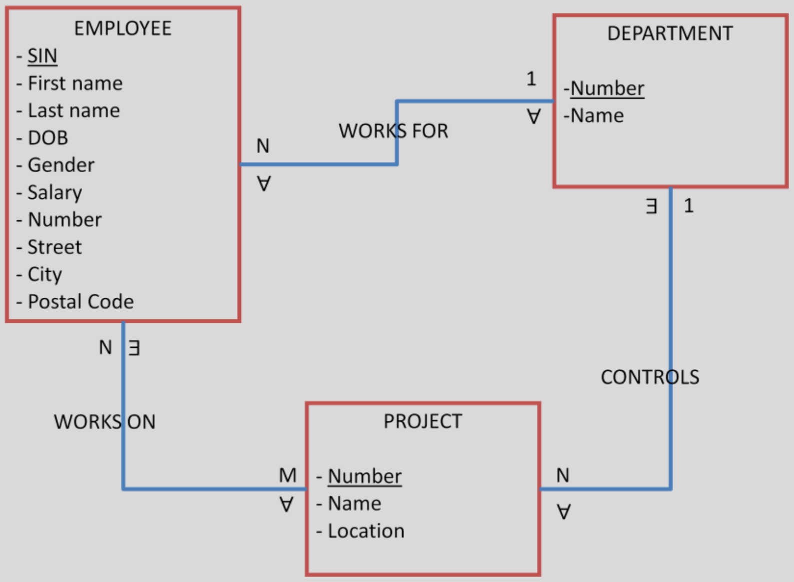
The WORKS ON edge is labeled with M and N, indicating a many-to-many relationship type: an employee can work on many projects and a project can have many employees working on it. Employee x can be working on 2 projects, say and project y can have 12 employees working on it. We choose to use different symbols on each side of the edge (M and N, as opposed to using N on both ends) so that we do not give the impression that the numbers must be equal on both sides (If employee x works 2 projects, say, we should not imply that project y must have 2 employees working on it).

Relationship Type Degrees: The degree of a relationship type is the number of entity types it relates. WORKS_ON, WORKS_FOR, and CONTROLS are all binary relationship type. Each relates two entity types. The degree of such a relationship is two. Relationship types could be of any degree, relating as many entity types as needed. Relationship types of very high degrees are seldom used in practice.

Participation Levels: It is often helpful to include further information on the edges of the ERD, such as how each entity participates in the relationship. Does every employee have to work on a project? Does every department control some project? There are two levels of participation in a relationship, universal or full and existential or partial.

EXAMPLE 5– 203 Company ERD

The following ERD contains extra labels, in the form of existential and universal quantification symbols.



The EMPLOYEE side of WORKS FOR is labeled with ∀, indicating that every employee must work for some department. Similarly, the DEPARTMENT side of WORKS FOR is labeled with ∀, indicating that every department must have some employee working for it. In other words, EMPLOYEE fully participates in WORKS FOR and DEPARTMENT fully participates in WORKS FOR.

The EMPLOYEE side of WORKS ON is labeled with ∃, indicating that some (but not necessarily all) employees work on projects. However, PROJECT's side of WORKS ON is labeled with ∀, emphasizing that every project must have some employee working on it.

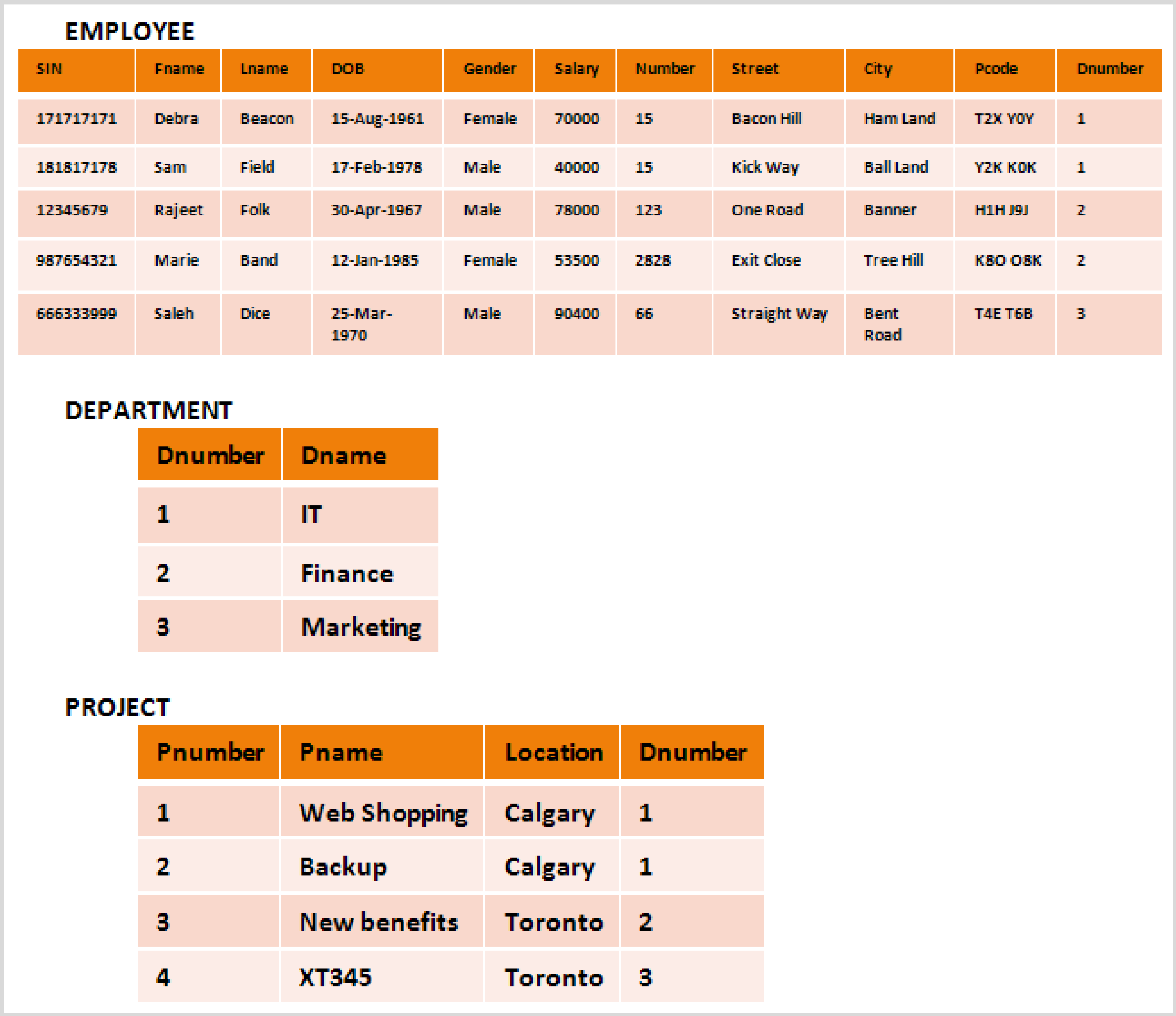
The DEPARTMENT participation in CONTROLS is partial, emphasizing that not all departments need to have projects under their control. However, projects must be always controlled by some departments.

II– Mapping ERD to a Schema

The ERD is one step towards designing a database. It specifies how data is related. In a database, information is specified in the form of tables. The structure of these tables is called the *database schema*. We will present an algorithm that translates an ERD to a database schema.

EXAMPLE 6 – Databases and Schema

The following shows an example database that contains three tables: EMPLOYEE, DEPARTMENT, and PROJECT.



EMPLOYEE

SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode	Dnumber
171717171	Debra	Beacon	15-Aug-1961	Female	70000	15	Bacon Hill	Ham Land	T2X Y0Y	1
181817178	Sam	Field	17-Feb-1978	Male	40000	15	Kick Way	Ball Land	Y2K K0K	1
12345679	Rajeet	Folk	30-Apr-1967	Male	78000	123	One Road	Banner	H1H J9J	2
987654321	Marie	Band	12-Jan-1985	Female	53500	2828	Exit Close	Tree Hill	K8O O8K	2
666333999	Saleh	Dice	25-Mar-1970	Male	90400	66	Straight Way	Bent Road	T4E T6B	3

DEPARTMENT

Dnumber	Dname
1	IT
2	Finance
3	Marketing

PROJECT

Pnumber	Pname	Location	Dnumber
1	Web Shopping	Calgary	1
2	Backup	Calgary	1
3	New benefits	Toronto	2
4	XT345	Toronto	3

The database schema for this example is simply:

EMPLOYEE

SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode	Dumber

DEPARTMENT

Dnumber	Dname

PROJECT

Pnumber	Pname	Location	Dnumber

Mapping algorithm: The input to the mapping algorithm is an ERD and the output is a database schema. In this version, we ignore one-to-one relationship types, but we will discuss how to handle them later.

1. Each entity type is translated to a table; its attributes become columns
2. Each many-to-many relationship type becomes a table; the columns are the primary keys of the participating entity types
3. For each one-to-many relationship type, add the primary keys of the entity type on the one side as columns in the table corresponding to the entity type on the many side

EXAMPLE 7 – Mapping

The schema corresponding to the ERD in Example 4 is as follows:

The EMPLOYEE table has all the attributes of the EMPLOYEE entity type as columns. In addition it has the Number of the DEPARTMENT entity type induced by the WORKS FOR relationship type. Since this is a one-to-many relationship and DEPARTMENT is on the one side, the primary key of DEPARTMENT has been duplicated in the EMPLOYEE table, Dnumber. Since Dnumber came from another place, it is called a *foreign key*.

EMPLOYEE

SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode	Dnumber

The DEPARTMENT table has the two attributes of the DEPARTMENT entity type..

DEPARTMENT

Dnumber	Dname

The PROJECT table has the three attributes of the PROJECT entity type. In addition, it has the foreign key Dnumber from the DEPARTMENT entity type. This is a result of the CONTROLS relationship. Since DEPARTMENT is on the one side, DEPARTMENT's primary key (DNumber) is duplicated in the table PROJECT.

PROJECT

Pnumber	Pname	Location	Dnumber

The only many-to-many relationship type is WORKS ON. A new table is created as a result of this relationship, we called it PROJ_EMP. This table has the primary keys of the participating entity types, namely PROJECT and EMPLOYEE.

PROJ_EMP

<u>SIN</u>	<u>Pnumber</u>

Notice that we have renamed number to Dnumber (for department) and Pnumber (for project) to avoid any confusion whether we are referring to the number of the project or the department. The same also applies to name (Dname and Pname)

EXAMPLE 8 – An instance database

The database of Example 6 forms an instance of the schema generated in Example 7, after the addition to the following able

PROJ_EMP

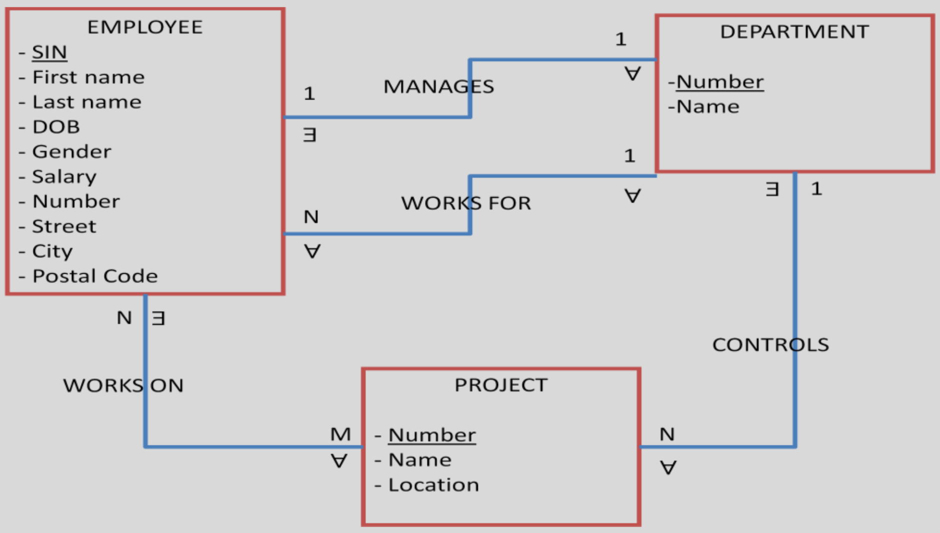
SIN	Pnumber
171717171	1
171717171	2
181817178	1
181817178	2
123456789	3
666333999	4

One-to-one relationship types: This kind of relationships is rare in databases. Nevertheless, they can occur. For instance, a MANAGES relationship type could be relating EMPLOYEE and DEPARTMENT. This relationship captures which employee manages which department. Since a department can have exactly one manager and an employee can manage only one department this would be a one-to-one relationship.

One-to-one relationships are treated like one-to-many relationships. They do not cause new tables to be created. Instead they augment the structure of existing tables. With one-to-many relationships, the primary key on the "one" side of the relationship is duplicated as a foreign key at the "many" side of the relationship. One-to-one relationships are a little bit more difficult since both ends of a relationship type are the "one" end. To guarantee a good design, we have to take participation into consideration. If both sides (entity types) participate in the relationship at the same level (both are partial or both are full), then it does not matter which primary key is duplicated in the other end as a foreign key. It is your pick. However, if one entity type participates partially and the other fully, we take the primary key of the partial side and duplicates it as a foreign key in table corresponding to the full side.

EXAMPLE 9 – One-to-one relationships

Consider the following updated ERD. We have added a one-to-one MANAGES relationship, between DEPARTMENT and EMPLOYEE.



This would cause the schema developed in Example 7 to change. The DEPARTMENT table will be as follows:

DEPARTMENT

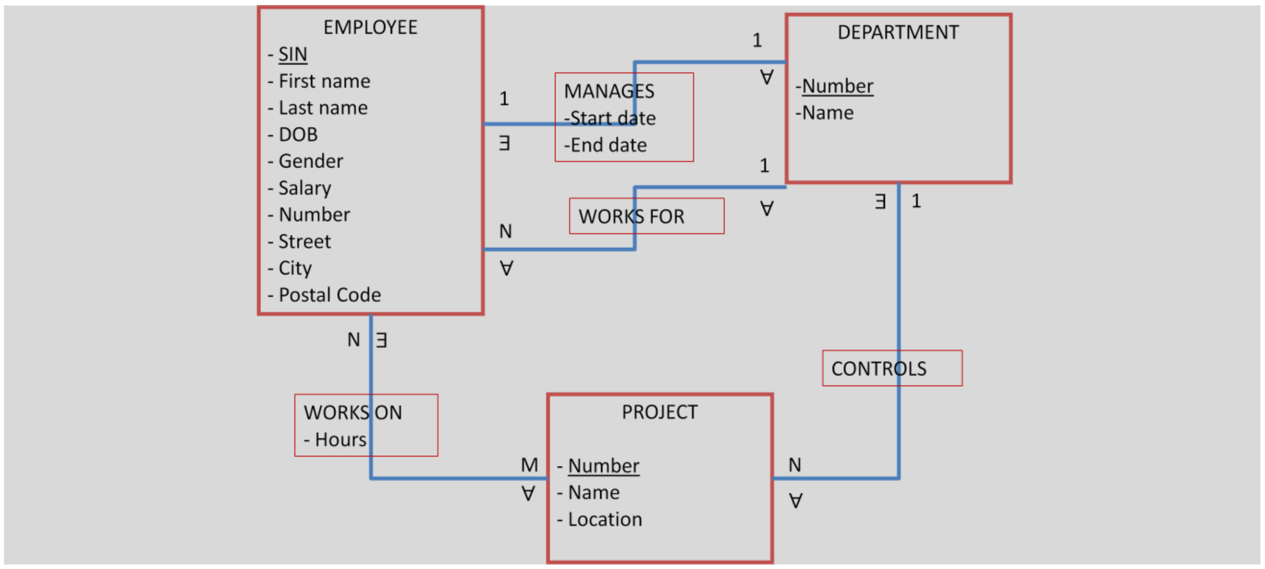
Dnumber	Dname	MGR_SIN

We have added the SIN of the EMPLOYEE who manages a DEPARTMENT, as column MGR_SIN. EMPLOYEE is on the partial participation side of MANAGES and DEPARTMENT is on the full side (not all employees are manages, but all departments must have managers). So, we have duplicated EMPLOYEE's primary key in the DEPARTMENT table.

Attributes for relationship types: We have given only entity types attributes, but relationship types can also have attributes. Consider for instance the WORKS ON relationship. It simply tells us which employee is working on which project, but it does not tell us how many hours the employee is working on each project. To capture this extra information, the WORKS ON relationship type must have an attribute, Hours. As another example, consider the MANAGES relationship type of Example 9. Here also, we may need to know the effective dates of a manager. If this is the case, then we need two attributes of the MANAGES relationship type: Start date and End date.

EXAMPLE 10 – Relationship attributes

In the following ERD, we have added attributes to MANAGES and WORKS ON, as we have discussed earlier.



Now, we are ready to refine the algorithm for mapping from ERD to database schema so that it takes into consideration one-to-one relationships and relationship type attributes.

Complete Mapping Algorithm:

1. Each entity type E is translated to a table T (to emphasize that T is a result of E, we write T(E))
   - T's columns are E's attributes
2. Each many-to-many relationship type R, relating entity types E1 and E2, becomes a table T
   - T's columns are R's attributes
   - the primary key of E1 and E2 is added as columns in T
3. For each one-to-many relationship type R, relating E1 to E2 with E1 on the "one" side:
   - add the primary key of E1 as columns in T(E2)
   - any attributes that R has become columns in T(E2)
4. For each one-to-one relationship type R, relating E1 to E2 with E1 on a partial participation side or both E1 and E2 fully participate in R:
   - add the primary key of E1 as columns in T(E2)
   - any attributes that R has become columns in T(E2)

EXAMPLE 11 – Mapping

We apply the Complete Mapping Algorithm to the ERD in Example 10, step by step.

Step 1 results in the following three tables, corresponding to the three entity types in the ERD:

EMPLOYEE

SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode

DEPARTMENT

Dnumber	Dname

PROJECT

Pnumber	Pname	Location

Step 2 results in adding a fourth table corresponding to the only many-to-many relationship type in the ERD. Hours is an attribute of WORKS ON, but SIN and Pnumber are foreign keys.

PROJ_EMP

SIN	Pnumber	Hours

The remaining relationship types are either one-to-one or one-to-many. They do not cause the creation of new tables. Instead, the augment the structures of the already existing four tables by adding more columns as foreign keys.

Step 3 with the WORKS FOR relationship results in the following change for table EMPLOYEE

EMPLOYEE

SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode	Dnumber

Step 3 with the CONTROLS relationship results in the following change for table PROJECT:

PROJECT

Pnumber	Pname	Location	Dnumber

Step 4 with the MANAGES relationship results in the following change for table DEPARTMENT:

DEPARTMENT

Dnumber	Dname	MGR_SIN	StartDate	EndDate

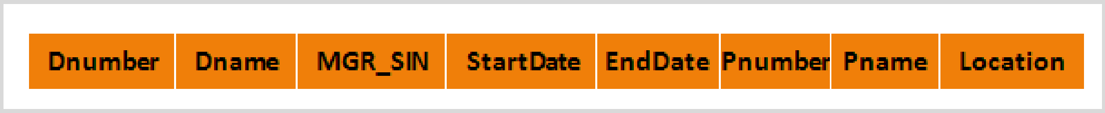
III– Design Principles

What makes a good design? Why did we copy the primary key from the one side to be duplicated as a foreign key in the table corresponding to the many side? What happens if we do the opposite? There are three basic design principles in databases.

1. Meaning of a schema should be easily explained: Do not combine attributes from different entity types into a single table.

EXAMPLE 12 – Schema meaning

We could have combined both PROJECT and DEPARTMENT into one schema



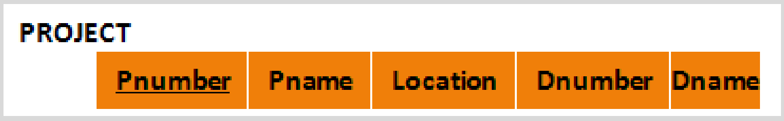
Dnumber	Dname	MGR_SIN	StartDate	EndDate	Pnumber	Pname	Location

The resulting table would have an unclear meaning.

2. Reduce redundancy: Unnecessary redundancy can lead to *modification anomaly.* Such anomalies arise when modification of data is required.

EXAMPLE 13 – Modification anomaly

Assume that we have designed the project schema as follows:



PROJECT

<u>Pnumber</u>	Pname	Location	Dnumber	Dname

The replication of Dnumber in project is unnecessary. A modification anomaly arises if the 203 company decides to change the name of the IT department, say. Such un update will not be limited to the DEPARTMENT table, but it will need to be done in PROJECT and other places where Dnumber is repeated.

Note that in our original design of Example 11, Dname occurs in one place only, the DEPARTMENT table, and any modification to it will be limited to the DEPARTMENT table. The remaining table will remain untouched.

Step 3 of the Complete Mapping Algorithm instructs us to add the primary key of the "one" side in the table corresponding to the "many" side. If we do the opposite, it will result in unnecessary redundancy.

EXAMPLE 14 – Unnecessary redundancy

Let's consider the CONTROLS relationship type. Assume instead of following our mapping algorithm we copy the primary key of PROJECT in DEPARTMENT (the algorithm asks us to do the opposite). This would result in the following schema

PROJECT

Pnumber	Pname	Location

DEPARTMENT

Dnumber	Dname	MGR_SIN	Start date	End date	Pnumber

Our example database would contain

PROJECT

Pnumber	Pname	Location
1	Web Shopping	Calgary
2	Backup	Calgary
3	New benefits	Toronto
4	XT345	Toronto

DEPARTMENT

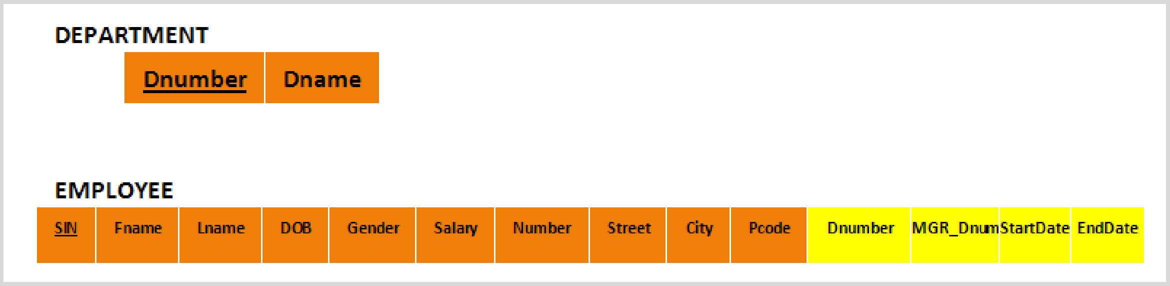
Dnumber	Dname	MGR_SIN	StartDate	EndDate	Pnumber
1	IT	171717171	12-Feb-2008	NUL	1
1	IT	171717171	12-Feb-2008	NUL	2
2	Finance	123456789	1-Mar-2002	NUL	3
3	Marketing	666333999	1-Jan-2005	NUL	4

Since the IT department manages two projects, this results in two tuples corresponding to the IT department in DEPARTMENT. In general, this could result in much more redundancy, since it requires a department row for each project it manages.

3. Reduce NUL values: NUL values are blank values and are unavoidable, but the design should try to minimize them as much as possible. NUL values waist space. In the DEPARTMENT table the EndDate for the managers is unknown; we have to leave them blank until the manager completes the managerial duties. Also NUL values result in confusion. What does a NUL value mean: not applicable, unknown, or to be recorded? It is not always possible to determine the meaning of a NUL value.

EXAMPLE 15 – Unnecessary redundancy

Let's consider the MANAGES relationship type. Assume instead of following our mapping algorithm we copy the primary key of DEPARTMENT in EMPLOYEE (the algorithm asks us to do the opposite). This would result in the following schema



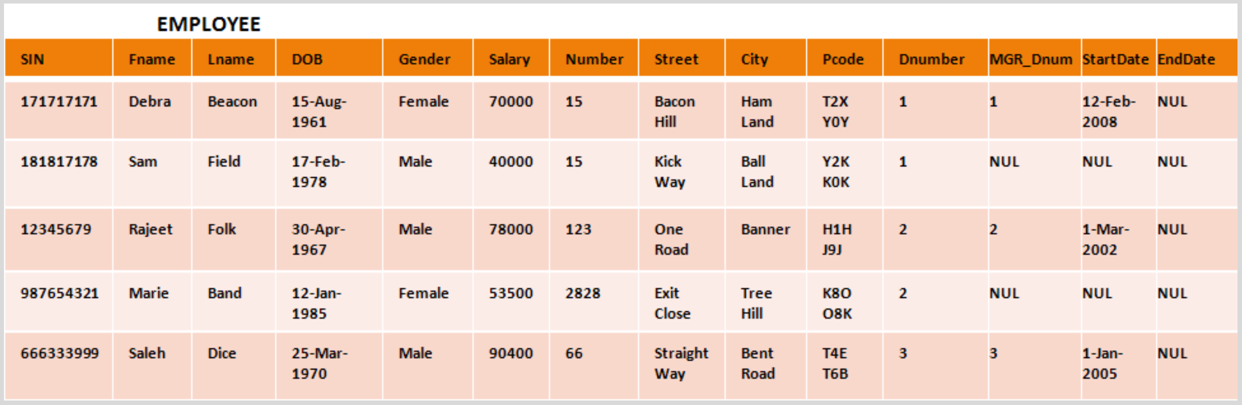
DEPARTMENT

Dnumber	Dname

EMPLOYEE

SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode	Dnumber	MGR_Dnum	StartDate	EndDate

Since not all employees are managers (EMPLOYEE's participation in MANAGES is partial), this would result in unnecessary NUL values in EMPLOYEE.



EMPLOYEE

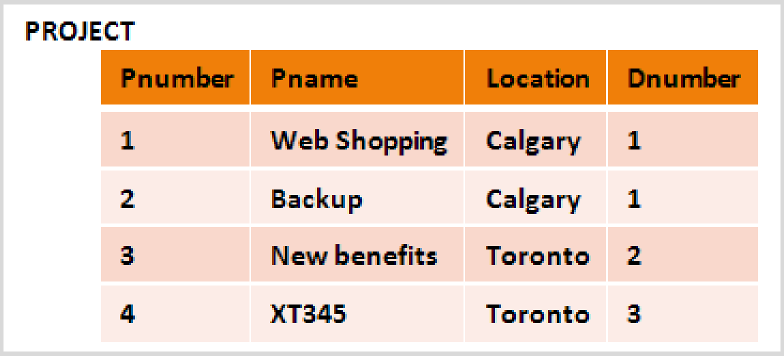
SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode	Dnumber	MGR_Dnum	StartDate	EndDate
171717171	Debra	Beacon	15-Aug-1961	Female	70000	15	Bacon Hill	Ham Land	T2X Y0Y	1	1	12-Feb-2008	NUL
181817178	Sam	Field	17-Feb-1978	Male	40000	15	Kick Way	Ball Land	Y2K K0K	1	NUL	NUL	NUL
12345679	Rajeet	Folk	30-Apr-1967	Male	78000	123	One Road	Banner	H1H J9J	2	2	1-Mar-2002	NUL
987654321	Marie	Band	12-Jan-1985	Female	53500	2828	Exit Close	Tree Hill	K8O O8K	2	NUL	NUL	NUL
666333999	Saleh	Dice	25-Mar-1970	Male	90400	66	Straight Way	Bent Road	T4E T6B	3	3	1-Jan-2005	NUL

IV– Tables and Relations

A database schema results in a collection of tables. That is a database is collection of tables containing the desired information. Tables are relations as defined in Chapter 1. Recall that a relation on sets A_1 , A_2 , … , A_n is a subset of A_1 x A_2 x … x A_n. In fact, most people call a table a relation (do not confuse this with relationships and relationship types)

EXAMPLE 16 – Tables as relations

Consider the project table



PROJECT

Pnumber	Pname	Location	Dnumber
1	Web Shopping	Calgary	1
2	Backup	Calgary	1
3	New benefits	Toronto	2
4	XT345	Toronto	3

This table can be represented in set notation as follows:
PROJECT = {(1, Web Shopping, Calgary, 1), (2, Backup, Calgary, 1), (3, New benefits, Toronto, 2), (4, XT345, Toronto, 3)

This is the reason database experts call a row a *tuple* because it simply is. It should be easily seen if
N is the set of natural numbers,
M is the set of names,
L is the set of locations,

PROJECT $\subseteq$ N×M×L×N

V– Querying the Database

The Structured Query Language (SQL) is an English-like computer language that is specific for databases. The language has two parts: the Data Definition Language (DDL) used to define the database such as the table structure and the Data Manipulation Language (DML) used to insert, delete, modify, and query data. In this course, we limit discussion to queries. A *query* is a question submitted to the database and the answer is a relation (a collection of tuples or rows).

Basic SQL: A basic SQL statement has two clauses SELECT and FROM. The FROM clause specifies which tables the query is being performed on and the SELECT clause specifies the columns that need to be returned.

Throughout this section, we use the following database example.

PROJECT

Pnumber	Pname	Location	Dnumber
1	Web Shopping	Calgary	1
2	Backup	Calgary	1
3	New benefits	Toronto	2
4	XT345	Toronto	3

DEPARTMENT

Dnumber	Dname	MGR_SIN	StartDate	EndDate
1	IT	171717171	12-Feb-2008	NUL
2	Finance	123456789	1-Mar-2002	NUL
3	Marketing	666333999	1-Jan-2005	NUL

EMPLOYEE

SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode	Dnumber
171717171	Debra	Beacon	15-Aug-1961	Female	70000	15	Bacon Hill	Ham Land	T2X Y0Y	1
181817178	Sam	Field	17-Feb-1978	Male	40000	15	Kick Way	Ball Land	Y2K K0K	1
12345679	Rajeet	Folk	30-Apr-1967	Male	78000	123	One Road	Banner	H1H J9J	2
987654321	Marie	Band	12-Jan-1985	Female	53500	2828	Exit Close	Tree Hill	K8O O8K	2
666333999	Saleh	Dice	25-Mar-1970	Male	90400	66	Straight Way	Bent Road	T4E T6B	3

PROJ_EMPLOYEE

SIN	Pnumber	Hours
171717171	1	15
171717171	2	25
181817178	1	30
181817178	2	10
123456789	3	40
666333999	4	40

EXAMPLE 17 – Projection in SQL

The following SQL statement projects PROJECT to two columns:

```
SELECT  Pname, Location
FROM    PROJECT
```

That is, the answer to this question is:

Pname	Location
Web Shopping	Calgary
Backup	Calgary
New Benefits	Toronto
XT345	Toronto

More useful SQL statements require a WHERE clause, which specifies a Boolean condition that filters the returned tuples. This operation is referred to by selection.

EXAMPLE 18 – Selection in SQL

The following SQL statement selects from PROJECT only those projects located in Calgary

```
SELECT  Pnumber, Pname, Location, Dnumber
FROM    PROJECT
WHERE Location = 'Calgary'
```

The result of this query is:

Pnumber	Pname	Location	Dnumber

1	Web Shopping	Calgary	1
2	Backup	Calgary	1

There is a cleaner way to write this same query. Instead of listing all the columns of STUDENT, we can simply use the wild card *****:

SELECT *
FROM PROJECT
WHERE Location = 'Calgary'

EXAMPLE 18 – Selection and Projection in SQL

Selection and projection can be combined together.

SELECT Pname, Location
FROM PROJECT
WHERE Location = 'Calgary'

The result of this query is:

Pname	Location
Web Shopping	Calgary
Backup	Calgary

Set operations in SQL: The result of a query is a relation, which is a set. So SQL allows us to combine the sets resulting from two different queries into one set using operations on sets (union, intersection, and difference). The general structure of such queries is:

Query 1
set operation
Query 2

Where *set operation* is any of UNION, INTERSECT (for intersection), or MINUS (for set difference)

EXAMPLE 19 – Set operations in SQL

The result of the following query is all projects that are located in Calgary or Toronto

SELECT *
FROM PROJECT
WHERE Location = 'Calgary'
UNION
SELECT *
FROM PROJECT
WHERE Location = 'Toronto'

The result of the following query is the SIN of all female managers

```
SELECT  SIN
FROM   EMPLOYEE
WHERE Gender = 'Female'
INTERSECT
SELECT  MGR_SIN
FROM   DEPARTMENT
```

The result of the following query is the SIN of all non-manager males

```
SELECT  SIN
FROM   EMPLOYEE
WHERE Gender = 'Male'
MINUS
SELECT  MGR_SIN
FROM   DEPARTMENT
```

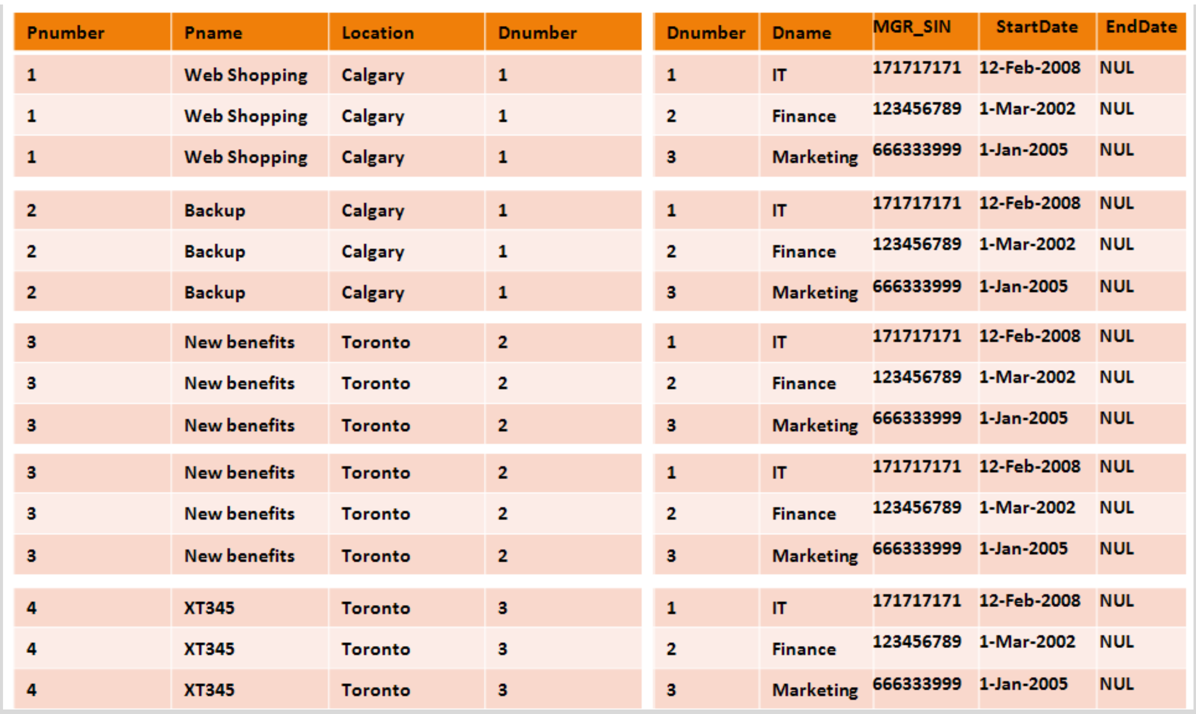
Natural Joins: PROJECT has the foreign key Dnumber which relates to DEPARTMENT. If we need to know which department controls the Web Shopping project, from table PROJECT it can be determined that is department number 1. Another query is required from table DEPARTMENT to figure out the rest of the department information such as the department name. SQL allows us to formulate such queries in one shot using natural joins.

EXAMPLE 20 – Cartesian products and joins

Consider the following query:

```
SELECT  *
FROM   PROJECT, DEPARTMENT
```

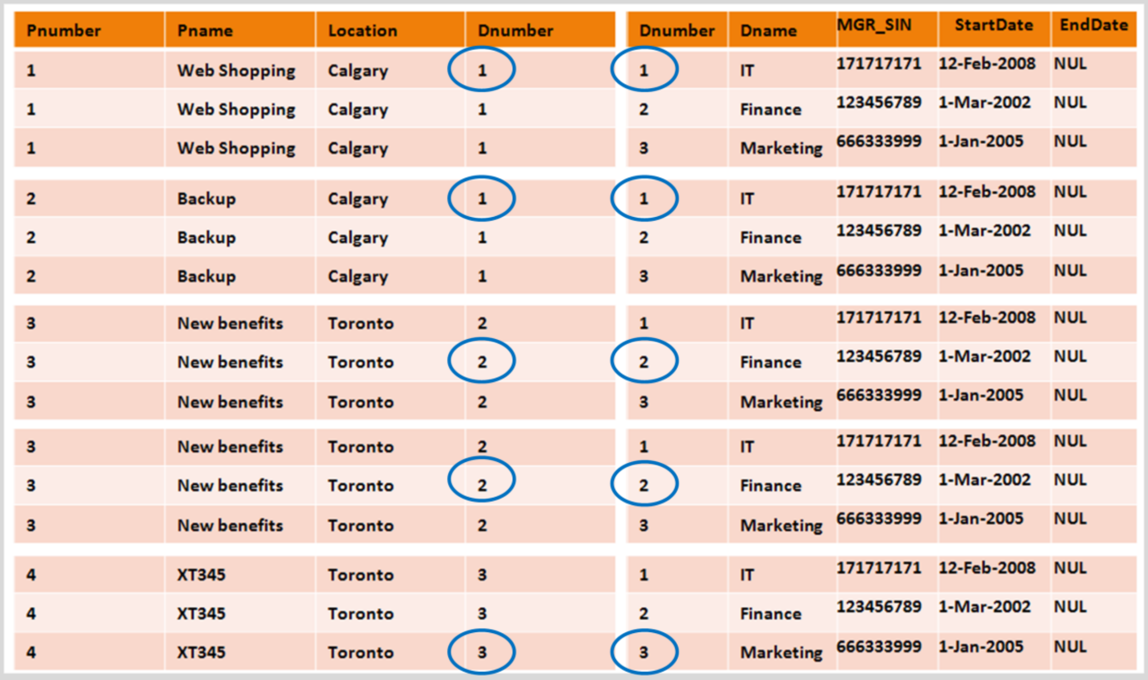
This query gives us the product of the set PROJECT and the set EMPLOYEE, or PROJECT × EMPLOYEE. That is, the result is:



Pnumber	Pname	Location	Dnumber	Dnumber	Dname	MGR_SIN	StartDate	EndDate
1	Web Shopping	Calgary	1	1	IT	171717171	12-Feb-2008	NUL
1	Web Shopping	Calgary	1	2	Finance	123456789	1-Mar-2002	NUL
1	Web Shopping	Calgary	1	3	Marketing	666333999	1-Jan-2005	NUL
2	Backup	Calgary	1	1	IT	171717171	12-Feb-2008	NUL
2	Backup	Calgary	1	2	Finance	123456789	1-Mar-2002	NUL
2	Backup	Calgary	1	3	Marketing	666333999	1-Jan-2005	NUL
3	New benefits	Toronto	2	1	IT	171717171	12-Feb-2008	NUL
3	New benefits	Toronto	2	2	Finance	123456789	1-Mar-2002	NUL
3	New benefits	Toronto	2	3	Marketing	666333999	1-Jan-2005	NUL
3	New benefits	Toronto	2	1	IT	171717171	12-Feb-2008	NUL
3	New benefits	Toronto	2	2	Finance	123456789	1-Mar-2002	NUL
3	New benefits	Toronto	2	3	Marketing	666333999	1-Jan-2005	NUL
4	XT345	Toronto	3	1	IT	171717171	12-Feb-2008	NUL
4	XT345	Toronto	3	2	Finance	123456789	1-Mar-2002	NUL
4	XT345	Toronto	3	3	Marketing	666333999	1-Jan-2005	NUL

It pairs each row of PROJECT with every row of DEPARTMENT

A query like this is not useful by itself. We need to filter the results further, leaving only the meaningful matches. For instance, it makes sense to only keep the IT department beside the Web Shopping project. That is we want the Dnumber from PROJECT to be the same as the Dnumber from DEPARTMENT. The matching values are encircled:

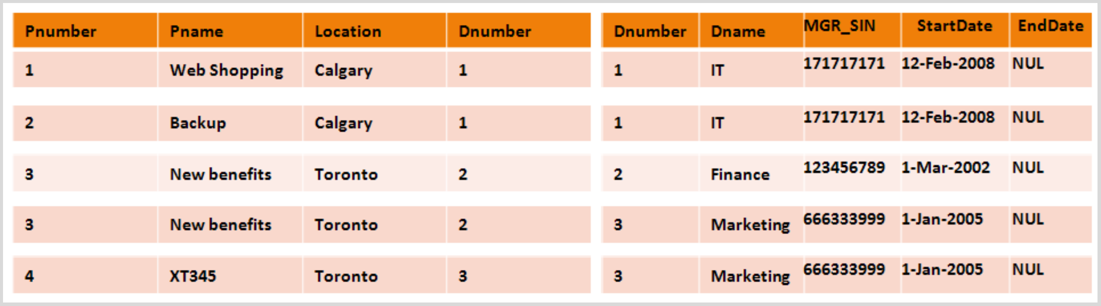


Pnumber	Pname	Location	Dnumber	Dnumber	Dname	MGR_SIN	StartDate	EndDate
1	Web Shopping	Calgary	1	1	IT	171717171	12-Feb-2008	NUL
1	Web Shopping	Calgary	1	2	Finance	123456789	1-Mar-2002	NUL
1	Web Shopping	Calgary	1	3	Marketing	666333999	1-Jan-2005	NUL
2	Backup	Calgary	1	1	IT	171717171	12-Feb-2008	NUL
2	Backup	Calgary	1	2	Finance	123456789	1-Mar-2002	NUL
2	Backup	Calgary	1	3	Marketing	666333999	1-Jan-2005	NUL
3	New benefits	Toronto	2	1	IT	171717171	12-Feb-2008	NUL
3	New benefits	Toronto	2	2	Finance	123456789	1-Mar-2002	NUL
3	New benefits	Toronto	2	3	Marketing	666333999	1-Jan-2005	NUL
3	New benefits	Toronto	2	1	IT	171717171	12-Feb-2008	NUL
3	New benefits	Toronto	2	2	Finance	123456789	1-Mar-2002	NUL
3	New benefits	Toronto	2	3	Marketing	666333999	1-Jan-2005	NUL
4	XT345	Toronto	3	1	IT	171717171	12-Feb-2008	NUL
4	XT345	Toronto	3	2	Finance	123456789	1-Mar-2002	NUL
4	XT345	Toronto	3	3	Marketing	666333999	1-Jan-2005	NUL

To eliminate the rest of the rows, we add a WHERE clause to filter these rows.

SELECT *
FROM PROJECT, DEPARTMENT
WHERE PROJECT.Dnumber = DEPARTMENT.Dnumber

This is called a (natural) join and the result of the query is:



Pnumber	Pname	Location	Dnumber	Dnumber	Dname	MGR_SIN	StartDate	EndDate
1	Web Shopping	Calgary	1	1	IT	171717171	12-Feb-2008	NUL
2	Backup	Calgary	1	1	IT	171717171	12-Feb-2008	NUL
3	New benefits	Toronto	2	2	Finance	123456789	1-Mar-2002	NUL
3	New benefits	Toronto	2	3	Marketing	666333999	1-Jan-2005	NUL
4	XT345	Toronto	3	3	Marketing	666333999	1-Jan-2005	NUL

Complex Boolean conditions: The WHERE clause can include complex Boolean conditions using AND, OR, and NOT. The comparison operators used in SQL are:

- Equals =
- Greater than >
- Less than <
- Different (not equal) <> or !=
- Less than or equal <=
- Greater than or equal >=

EXAMPLE 21 – More SQL examples

The following query retrieves the last names and DOBs of female employees whose salary is more the 40K

SELECT Lname, DOB
FROM EMPLOYEE
WHERE Gender = 'Female'
AND Salary > 40000

The following query retrieves the SIN, last name, and first name of employees whose salary is between (inclusive) 30K and 50K

SELECT SIN, Lname, Fname
FROM EMPLOYEE
WHERE Salary >= 30000
AND Salary <= 50000

The following query retrieves the SIN, last and first names of all male employees who earn more than 30K and all female employees who earn above 40K

```
SELECT  SIN, Lname, Fname
FROM    EMPLOYEE
WHERE (Gender = 'Male' AND Salary > 30000)
OR      (Gender = 'Female' AND Salary > 40000)
```

Alternatively, the same query can be expressed using set union as follows:

```
SELECT  SIN, Lname, Fname
FROM    EMPLOYEE
WHERE (Gender = 'Male' AND Salary > 30000)
UNION
SELECT  SIN, Lname, Fname
FROM    EMPLOYEE
WHERE (Gender = 'Female' AND Salary > 40000)
```

The following query retrieves the addresses of all employees who work for the IT department:

```
SELECT Number, Street, City, Pcode
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.Dnumber = DEPARTMENT.Dnumber
AND Dname = 'IT'
```

Building a query trace: Tracing SQL queries with join conditions can be tedious. Consider the following query:

```
SELECT SIN, Number, Street, City, Pcode, Dname
FROM EMPLOYEE, DEPARTMENT
WHERE EMPLOYEE.Dnumber = DEPARTMENT.Dnumber
AND Dname = 'Marketing'
AND Salary > 70000
```

One straightforward way is to start the trace by considering the Cartesian product of EMPLYEE and DEPARTMENT. If EMPLOYEE has *n* tuples and DEPARTMENT has *m* tuples, we would be starting with a result set of *n*×*m* tuples. In our case, this is 6×3 = 18 tuples, but in general this could be prohibitively large. The last two lines of the SQL query can restrict the result set since we need not consider all departments and all employees. There is exactly one department with Dname='Marketing' and we only have two employees whose salary > 70K. So, we need only work with 2×1= 2 tuples to construct the result of this query. So, we build the trace starting by applying the filtering imposed by the last two lines.

The table DEPARTMENT when filtered by **Dname = 'Marketing'**, leaves us with the following subset of DEPARTMENT:

Dnumber	Dname
3	Marketing

Similarly, applying the condition **Salary > 70000** gives us the following subset of EMPLOYEE:

SIN	Fname	Lname	DOB	Gender	Salary	Number	Street	City	Pcode	Dnumber
12345679	Rajeet	Folk	30-Apr-1967	Male	78000	123	One Road	Banner	H1H J9J	2
666333999	Saleh	Dice	25-Mar-1970	Male	90400	66	Straight Way	Bent Road	T4E T6B	3

The Cartesian product is now simpler to work with:

SIN	…	Salary	Number	Street	City	Pcode	Dnumber	Dnumber	Dname
12345679	…	78000	123	One Road	Banner	H1H J9J	2	3	Marketing
666333999	…	90400	66	Straight Way	Bent Road	T4E T6B	3	3	MArketing

Quickly we can infer that the result of this query is simply:

SIN	Number	Street	City	Pcode
666333999	66	Straight Way	Bent Road	T4E T6B

Ordering the query results: The rows returned as a result for a query are ordered in the way they are stored in the database. We can instruct the SQL query to order the result using an ORDER BY clause.

EXAMPLE 22 – ORDER BY clause

The following query orders the result by last name

SELECT Lname
FROM EMPLOYEE
WHERE Fname != 'Sarah'
ORDER BY Lname

The following query orders the result by last name in descending order

SELECT Lname
FROM EMPLOYEE
WHERE Fname != 'Sarah'
ORDER BY Lname DESC

The following query orders the result by last name and first name (if two employees have the same last name the tie is broken by their first names)

SELECT Lname , Fname, Salary
FROM EMPLOYEE

```
WHERE Fname != ‘Sarah’
ORDER BY Lname, Fname
```

Aggregate functions: Some of the aggregate (or group) functions in SQL are:

- SUM: sum of values in a column
- AVG: average of values in a column
- COUNT: number of not null values
- MIN: minimum value in a column
- MAX : maximum value in a column

EXAMPLE 23 – Aggregate functions

The following query determines the sum of all salaries

```
SELECT SUM(Salary)
FROM EMPLOYEE
```

The following query determines the number of records in EMPLOYEE

```
SELECT COUNT(*)
FROM EMPLOYEE
```

The following query determines the average salary in department number 1

```
SELECT AVG(Salary)
FROM EMPLOYEE
WHERE Dnumber = 1
```

The following query determines the minimum salary for female employees

```
SELECT MIN(Salary)
FROM EMPLOYEE
WHERE Gender = ‘Female’
```

The following query determines the maximum salary for male employees

```
SELECT MAX(Salary)
FROM EMPLOYEE
WHERE Gender = ‘Male’
```

GROUP BY clause: The GROUP BY allows us to apply aggregate functions for separate groups of values, such as counting the number of employees in each department.

EXAMPLE 24 – Aggregate functions

The following query determines the sum of all salaries for each department

```
SELECT Dnumber, SUM(Salary)
FROM EMPLOYEE
GROUP BY Dnumber
```

The result would be:

```
Dnumber         SUM(Salary)
1               11000
2               131500
3               90400
```

The following query calculates the average salary per gender

```
SELECT Gender, AVG(Salary)
FROM EMPLOYEE
GROUP BY Gender
```

HAVING clause: We can further filter the result of a group function by requiring the aggregate value to satisfy certain Boolean condition.

EXAMPLE 25 – Aggregate functions

The following query determines the number of projects each employee works on

```
SELECT SIN, COUNT(*)
FROM EMP_PROJ
GROUP BY SIN
```

The following query further filters the result to include only those employees who work on more than one project:

```
SELECT SIN, COUNT(*)
FROM EMP_PROJ
GROUP BY SIN
HAVING COUNT(*) > 1
```

Exercises

1. Design a simple **university** ER
    a. The university has students, courses, and instructors
    b. The database will be used to keep track of student completion of courses and the respective final letter grades
    c. We should be able to tell which instructor taught a given student and in what course
2. Design a simple **bookshop** ER
    - They carry only Books
    - They need to be able to search for a book by author, publisher, subject, etc..
    - They also record size, cost price, sale price, acquire date of each book
3. Design a simple **music library** ER
    - The library has all kinds of records (Disks, phonographs, etc…)
    - For each song/instrumental, the library keeps track of all musicians who are in making the song/instrumental (who did what: vocal, piano, drums, lyrics, composing, etc …)
    - It also keeps track duration, date and other properties of each song/instrumental
4. Map the ERD of questions 1, 2, and 3 to a database schema
5. On the example database, write the following queries in SQL:
    a. Retrieve employees whose salary is less than 30K
    b. Retrieve employees whose salary is less than 30K and live in Ham Land
    c. Retrieve employees whose salary is more than 30K or do not live in Ham Land
    d. Retrieve projects that are controlled by the finance department
    e. (very challenging) retrieve the departments that are control at least one project and are managed by a female employee
    f. Find the total number of hours each employee is working on projects
    g. Find the average hours that employees are working on projects
    h. Find the total salary per department, as long as the total is more than 100K

Chapter 6
Mingling Computers: Networking

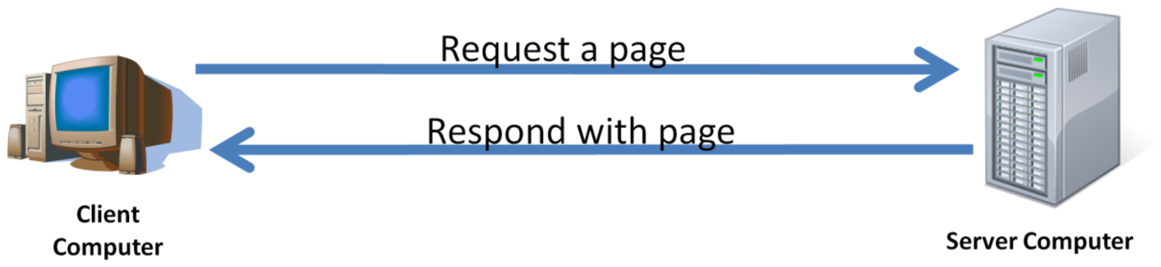
Computers are no longer computing machines only. They have become a communication tool. Browsing, instant messaging, voice over IP, and media streaming have become an integral part of our lives. But, how are actually things done? What happens behind the scene? Do you still think that the WWW and the Internet are the same thing? In this chapter, we focus our spot light on these and similar questions.

The chapter explains what the Internet and the WWW are and what rules computers need to follow so that they communicate with each other. Buzz terms such as TCP/IP, UDP, and HTTP will become meaningful. Communication cannot be made viable if it is not secure. The types of threats and the security mechanisms that are used to guard against threats are also discussed.

I – Basics of Networking

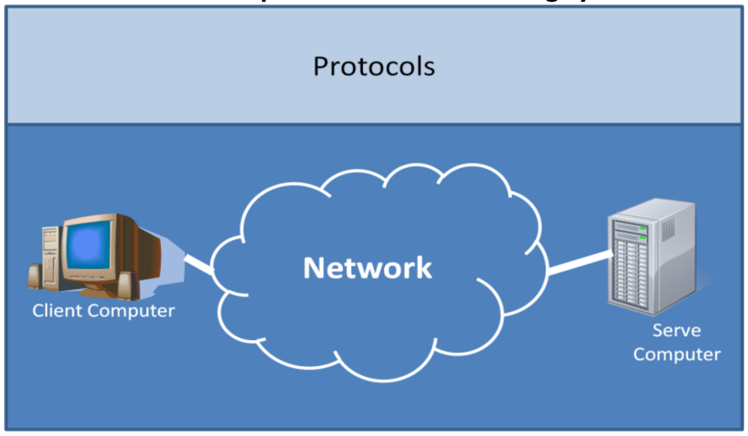
You open a browser and type in the address www.cbc.ca/index.html. Suddenly, you have a nicely presented Web page. What did actually happen? The page you have just opened was stored on a computer that is located somewhere else, possibly on the other side of the planet. The browser, running on your computer, sent a request to the computer that hosts the Web page you are requesting, and the latter sent the Web page to your browser. Let's accept this simplistic view for now. Your browser is called the *client* program and your computer is the *client computer*. The computer that is storing the Web pages is called the *server computer* and there is a program running on it whose job is to serve the client's requests; this program is the *server*.

Simplified view of client-server interaction



The Internet: For the client computer and the server computer to communicate, they have to be hooked up together through a *network*. A network is a collection of wires, wireless antennas and receivers, satellite connections, and any other mechanism that can make the two computers communicate. Yet, the network alone is not sufficient to establish communications. The mere fact that you and your friends have phones is not sufficient for both of you to make a telephone conversation. You need to know how to use the phone, you have to have a subscription with a telephone service provider, and you and your friend must be able to speak a common language. The rules that make it possible for computers to communicate are called *protocols*. A *networking system* consists of a network and protocols. The *Internet* is the largest networking system that ever existed.

The two components of a networking system



Internet Protocols: The Internet's protocol suite is called *TCP/IP*. IP stands for *Internet Protocol* and it mainly specifies an addressing mechanism. Computers need this addressing mechanism to locate each other. The telephone system has an addressing mechanism, which specifies the addresses of telephones: country code, area code, phone number. The postal system also has an addressing mechanism, without of which it would be impossible to send post cards to your friends. IP requires that each computer hooked to the Internet must have a unique IP address. The IP address consists of 4 positive integers separated by dots and each integer must be between 0 and 255. Work is in progress to enlarge the IP addresses so that more computers can be connected to the Internet.

EXAMPLE 1 – IP addresses

The following are valid IP addresses: 150.203.1.2, 127.15.30.12, and 57.83.77.90.

The following are not valid IP addresses: 150.256.1.2, 1.2.3, and 150.-1.34.56.

The server computer can have several server programs running on it. It is a large organization, just like your bank, where several servers with different specialties attend to customers. Typically bank branches have a main telephone number that connects you to the branch and then you will be asked to dial an extension number to connect to the right person that will serve you. The IP address is the address of the server computer. To connect to a specific server within the server computer, a client needs an extension number. These extension numbers are called *port numbers*. For instance, Web servers who serve Web pages are typically on port number 80 and email servers use port 25.

In addition to addressing, the IP protocol is also responsible for routing messages, but we will not dwell on routing here.

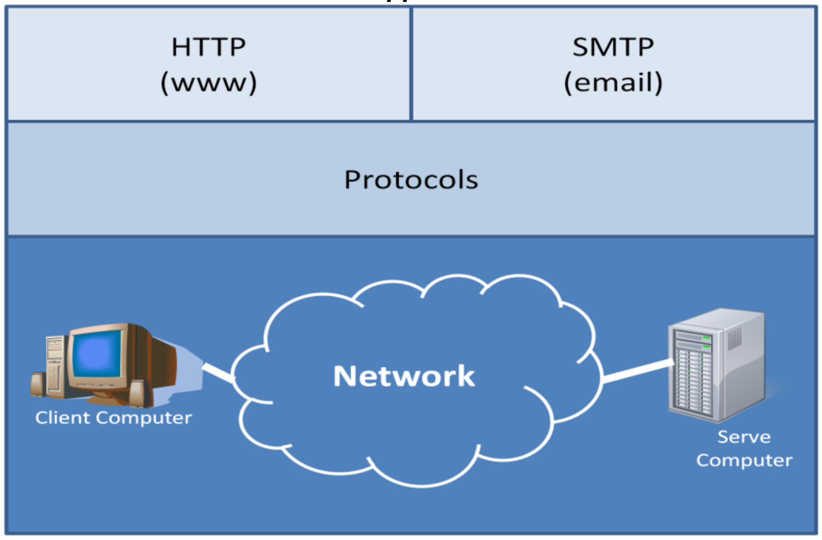
The first part of TCP/IP is TCP, which stands for *Transmission Control Protocol*. TCP specifies the rules for establishing connections between the client and server computers. These rules are:

1. The server must be listening to some port number waiting for clients' requests. This port number is the private extension for this server.

2. A client must request a connection with the desired server, specifying an IP address and a port number. This request is sent from the client computer to the server computer over the network.
3. The server accepts or rejects the request. If the connection request is accepted, a connection between the client and server is established.
4. The client can send its requests to the server once the connection is established and the server can send its responses back to the client.
5. Once the client has no further requests, the connection is closed by the client and the server.

WWW: The World Wide Web is the collection of HTML documents available through the Internet. The WWW is not the Internet and the Internet is not the WWW. The latter is simply an application built on top of the Internet, just like email is a different application built on the Internet. These applications rely on the protocols provided by the Internet, but they specify their own *application-level protocols*. When you communicate with your friends over the phone, you have to play by the rules set by the telephony system. Nevertheless on top of these protocols, you have your own (application-level) protocols. These could specify things such as do not call after midnight or before 10AM, or if you do not have enough credit, you miss call your friends so that they call you back. The protocol of WWW is called *HTTP*, which stands for *Hyper Text Transfer Protocol* and the email protocol is *SMTP* or *Simple Mail Transfer Protocol*.

WWW and email as applications of the Internet

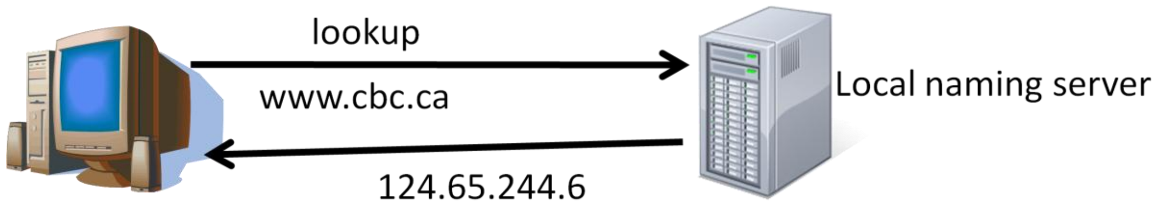


So, when you requested the page www.cbc.ca/index.html, the browser had to follow the 5 TCP steps outlined earlier. The missing link though is that the browser (the client) must request a connection with the Web server specifying an IP address and a port number. How was it able to lookup the IP address of the appropriate machine? First, we need to understand URLs.

URLs: A Universal Resource Locator is an easy to remember address given to a resource on the Web. The URL **http://www.cbc.ca/index.html** has several components. The **http://** prefix is specifying which protocol is being used, HTTP: the protocol of the Web. **www.cbc.ca** is the name of the server machine your client is trying to connect to. The last part **/index.html** is the name of a file stored on the server machine.

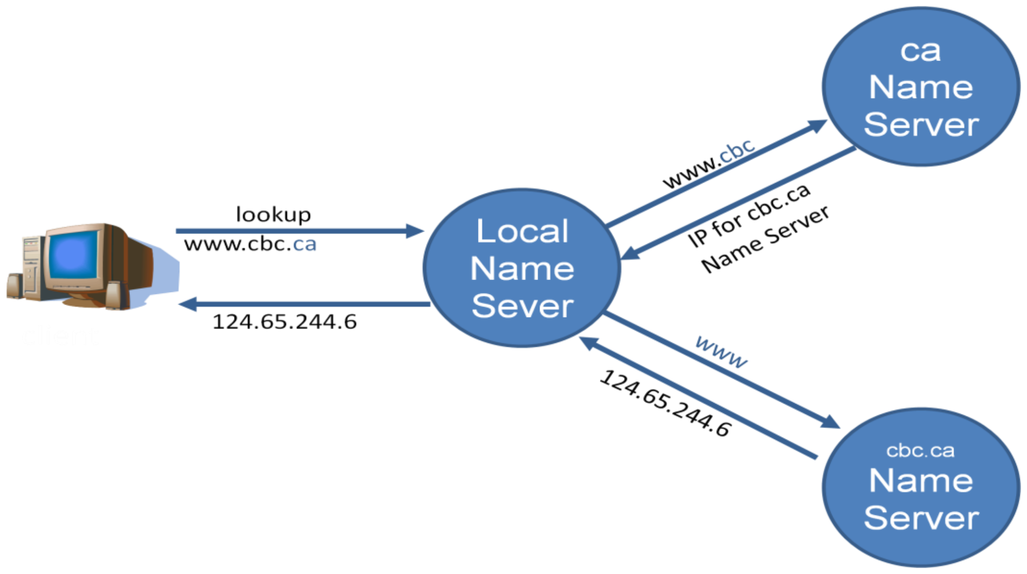
Naming service: Fortunately, one does not need to remember the IP address of the www.cbc.ca server. When someone requires a telephone number, typically they resort to the phone book to look it up. All what they need to remember is the name of the party they would like to dial up. In the Web, www.cbc.ca is the name of the server computer that one would like to communicate with. Your browser uses the equivalent of the telephone book to translate the symbolic name to an IP address. There are special servers that implement such a "phone book" for the Web. These are called *naming servers*. The browser communicates with a nearby naming server to lookup the IP for www.cbc.ca.

High-level view of the naming service



The implementation of the naming service is still more complicated than this simplistic explanation. Though it is feasible for every naming server to record the IPs of all the servers on the planet, this approach is not desirable. Every time a new server is added to the Internet, all such naming servers must be updated. This is infeasible. Instead, the local naming server communicates with other naming servers to resolve a name to an IP.

Illustration of name resolution



When the local naming server receives a lookup request for www.cbc.**ca** from the browser, it knows that it should contact the naming server that handles the **ca** domain. Anything that ends with **ca** must be registered with the **ca** naming server. The **ca** naming server examines the local naming server request and provides the local naming server with the IP for the naming server that handles **cbc.ca**. The local naming server sends another request to the **cbc.ca** naming server. **www** is typically the name of an actual computer registered with the **cbc.ca** naming server. So the latter supplies the IP to the local naming server which passes it back to the browser.

Note that in such an organization, the **cbc.ca** can add and remove server computers without having to update the **ca** naming server. The latter must be only updated if a new Web site is added under the ca domain. The browser can now request a connection with the required server. Note also that the naming servers make use of TCP/IP to communicate with each others as well.

HTTP: Now that the connection between the browser and the Web server are established, what kind of language do they speak with each other? They speak HTTP. That is, HTTP defines the structure and meaning of messages being exchanged between the browser and the Web server. For instance, the browser needs to be able to say: *get me the file index.html*. In what format does the server send the reply back? Keep in mind that all HTTP messages are sent using TCP/IP.

In HTTP there are two types of messages. *Requests* are sent from the client to the server and *responses* sent back from the server to the client. The two major request messages are called GET and POST. The GET message is generated every time a hyperlink is clicked. The POST request is generated when you fill in form data and press a submit button.

The general HTTP message structure is as follows:

General header
Additional headers
An empty line
An optional message

A GET request would look like:
GET /index.html HTTP/1.1

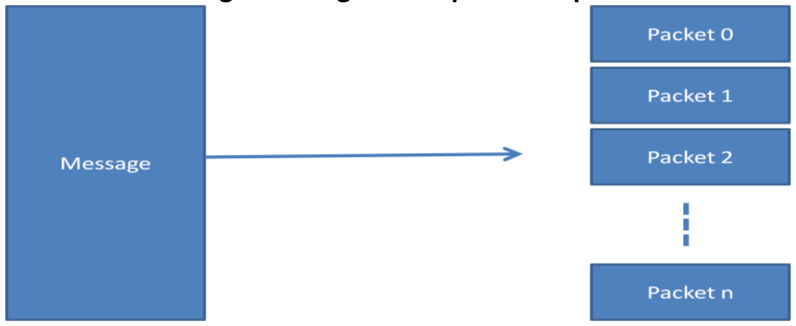
(must be followed by an empty line)
The last part of the request **HTTP/1.1** is specifying the protocol version.

A response message for this request may look like:
HTTP/1.1 200 OK
Date: Sat, 15 Nov 2009 14:55:11 GMT
Server: Apache/2.0.47
Accept-Ranges: bytes
Content-Length: 1107
Connection close
Content-Type: text/html; charset=ISO-8859-1

<HTML>
To the rest of the HTML page

UDP: TCP is not the only protocol used by the internet to transit information. The *Universal Datagram Protocol* (UDP) is less famous that TCP but is part of the Internet suite. A message, such as an HTTP response message including a an HTML file, is first broken down into smaller equal–sized messages called, *packets* or *datagrams*.

Breaking a message into equal-sized packets



These packets are sent separately over the network; they include extra information, such as the sender and receiver info and sequential numbers so that the original message can be re-assembled by the receiving party.

While TCP requires establishing a connection before exchanging messages, UDP does not. This makes TCP a *connection-oriented* protocol and UPD a *connectionless* protocol. The telephone system is a connection-oriented system. To talk to your friend, you have to establish a connection first: you have to dial your friend and she must pick up. The postal system is connectionless. To send your parents a post card, you do not have to have a connection established a head of time. You simply address the post card and affix a stamp on it and drop it in the postal office or a postal drop box, then you go away to do other things. The card gets routed in the postal system until it reaches its destination. Your parents need not be sitting there waiting for the card to arrive; they will be simply doing other errands too. UDP works much like the postal system. UDP is faster than TCP, but it is less reliable. Packets can be lost in UDP.

II – Secure Communication

When someone tries to login to their Internet banking application, a POST request that contains the login information is sent from the browser to the server hosting the application. What if someone manages to get this message and acquires your banking information? This is definitely possible if no extra measures are taken.

HTTPS: HTTP with security (HTTPS) enhances the HTTP protocol to make communication secure. It adds to HTTP a security protocol called *Secure Socket Layer* (SSL), which we will explain shortly. First, we need to understand the kinds of threats that exist with communication so that we know how to counter them.

Threat Types: There are three major types of threats that compromise communication:

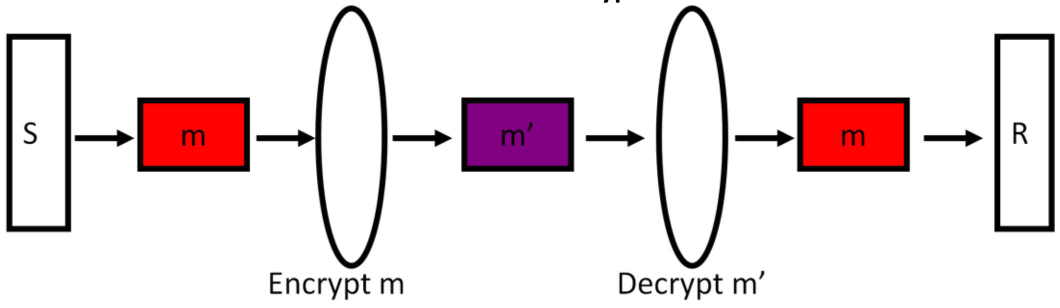
1. *Interception*: this type of threat takes place when a party obtains access to messages, data, or services that they are not authorized to access. If someone gets your login information, this is interception. Another example is if someone is able to break into the tax records and obtain your tax information.
2. *Modification*: this type takes place when an unauthorized change of messages, data, or services occurs. For instance, messages can be intercepted and changed before they arrive at their destination.
3. *Fabrication*: this threat refers to the unauthorized creation of messages, data, or services. Phishing email messages pretended to be sent from your bank to update your information, so that the phishers trap you to submit your confidential information to them is an example of fabrication threats.

Security Measures: To guard against these threats, two types of security measures can be taken.

1. *Encryption*: encode data and messages so that only intended parties can decode them.
2. *Authentication*: verify that the claimed identity by some party is authentic.

Encryption and authentication need to go hand-in-hand to create secure channels. A sender S encrypts a message m into a form no one other than the authorized parties can understand. The encrypted form of m, say m', is sent over the network. When the intended receiver R receives m', R will be able to decrypt m', generating m from it. m is called the *plain text* and m' is called the *cipher text.*

Illustration of encryption



Before we see how this guards against the three types of threats, let's pause and understand how encryption works. Consider the plain text:

1	2	3	4	5	6	7	8
C	P	S	C		2	0	3

which is an 8-charcter message. One (naïve) way to encrypt this message is to use a permutation key:

1	2	3	4	5	6	7	8
7	3	6	2	1	8	5	4

This key must be kept a secret and should be only shared by the sender and receiver. They key tells the sender how to encrypt the message and the receiver how to decrypt it. It simply tells the sender to shuffle the message in a certain way. Character 1 in the plain text goes to position 7 in the cipher text, character 2 of the plain text goes to 3 in the cipher text, and so on, generating the following cipher text:

1	2	3	4	5	6	7	8
0	S	2	P	C	3		C

The receiver can reverse this process easily and construct the plain text back from the cipher text.

While actual cryptosystems use similar ideas to this example, this approach is a very bad one. There are two differences between our naïve example and real systems. First, the encryption algorithm is far more complicated and second, encryption is done at the bit level rather than the character level. However, the example should serve the purpose of giving an idea how cryptosystems work.

Cryptosystems guard against the three types of threats listed earlier. An intruder will not be able to decrypt m' and therefore cannot get the message m. Also, an intruder cannot fabricate or modify the cipher text unless the secret key is compromised. However, encryption alone is insufficient to guarantee secure communication. If messages are being encrypted to guard against intruders, we also need to make sure that the identity of whom we are communicating with is indeed who we think they are. Authentication is as important as encryption.

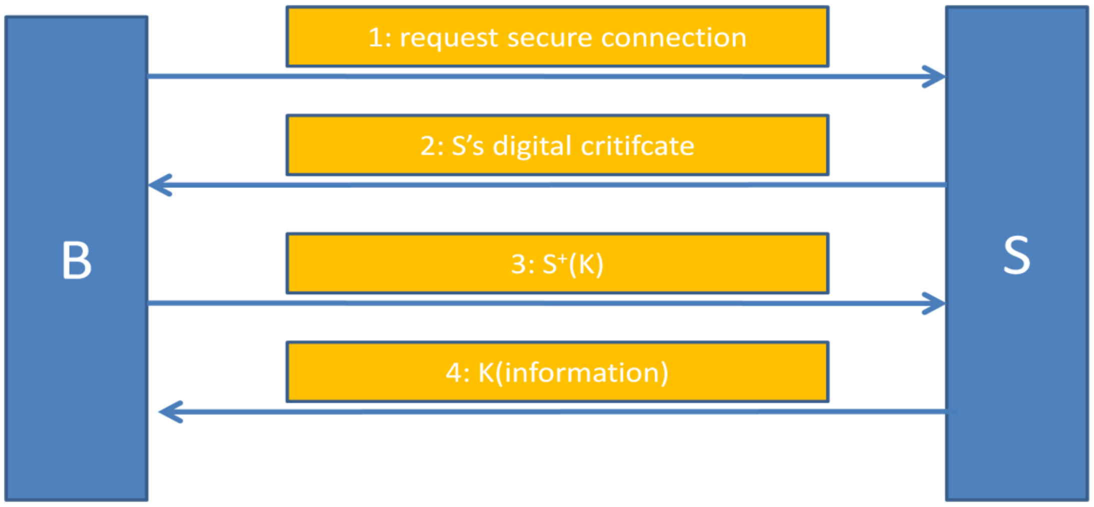
Types of cryptosystems: Encryption systems are of two kinds. In the *symmetric* type, parties share a secret key, which is used for encryption and decryption. These are called *secret* or *shared key* cryptosystems. The naïve permutation example that we have discussed earlier fall in this category. The second type is the *asymmetric* or *public key* cryptosystems. These make us of a pair of keys for each communicating party: a private key, which is always kept secret and is never shared with anyone, and a public key which is made available publicly. These keys are generated in such a way that they are reciprocal, yet it is not easy to construct a private key given the public one. Reciprocity means that if a message is encrypted using the public key, it can be only decrypted using the matching private key and vice versa.

To achieve encryption in an asymmetric system, the sender S makes use of the receiver's R public key to encrypt the message. R's private key is needed to decrypt the message and only R knows this key establishing secure communication.

To achieve authentication in such a system, the sender S uses its private key to encrypt the message. The receiver can authenticate the sender by applying R's public key to this message. If the message is recovered, it must be R who sent the message since only R knows its private key. Let X be a communicating party. Denote by X^+ and X^- X's public and private keys, respectively. Let m by the plain text. The message m,S^-(m) has two components, m itself, and an encryption of m using S's private key. The latter is called a *digital signature*. When R receives this message it applies S^+ to S^-(m) to decrypt the message, S^+ (S^-(m)) = m. R compares m from the first component of the message and the decrypted one of the second part. If they match, it must be the case the message is from S. In other words R authenticates S. If the message is confidential, S uses R's public key to encrypt the whole message. That is S sends R^+(m,S^-(m)) to ensure that only R gets the message.

SSL: The secure socket layer protocol (indicated by https in your browser) uses a combination of symmetric and asymmetric key cryptosystems to establish secure channels and to allow browsers to authenticate servers.

Illustrating SSL



The browser B requests a secure connection with the server S. Then S replies by sending its digital certificate to B, so that B authenticates S. Then, B generates a secret shared key K and sends it to S; but this key must remain a secret between B and S. So, B uses S's public key to encrypt K, S^+(K) . Now K can be used to encrypt the communication between R and S during this session. The key K will then be disposed off. Each time B requires a secure connection with S, a new disposable secret key is generated and used for one session only. Digital certificates are obtained through a third party for a fee.

Exercises

1. Which of the following is a legal IP address?
    a. 1.2.3
    b. 1.2.3.4
    c. 1000.2000.3.4
2. When would be possible for a locale name server to translate the name www.facebook.com without talking to any other naming server (.com or facebook.com)?
3. Which HTTP request is generated when a hyperlink is clicked?
4. Which HTTP request is generated when a form submit button is clicked?
5. Research the HEAD HTTP request and understand what it does.
6. Using the permutation key described in this chapter, determine the plain text from the following cipher text: **anmuay t**.