Programming Concepts

Topics

This course introduces basic programming concepts such as program structure, variable declaration, conditional and looping constructs, and the code/compile/run style of programming. This tutorial is intended for use as an introduction to these concepts for students who have no prior programming experience.

Introduction

Computer programs are collections of instructions that tell a computer how to interact with the user, interact with the computer hardware and process data. The first programmable computers required the programmers to write explicit instructions to directly manipulate the hardware of the computer. This “machine language” was very tedious to write by hand since even simple tasks such as printing some output on the screen require 10 or 20 machine language commands. Machine language is often referred to as a “low level language” since the code directly manipulates the hardware of the computer.

By contrast, higher level languages such as “C”, C++, Pascal, Cobol, Fortran, ADA and Java are called “compiled languages”. In a compiled language, the programmer writes more general instructions and a compiler (a special piece of software) automatically translates these high level instructions into machine language. The machine language is then executed by the computer. A large portion of software in use today is programmed in this fashion.

We can contrast compiled programming languages with interpreted programming languages. In an interpreted programming language, the statements that the programmer writes are interpreted as the program is running. This means they are translated into machine language on the fly and then execute as the program is running. Some popular interpreted languages include Basic, Visual Basic, Perl, Python, and shell scripting languages such as those found in the UNIX, Linux and MacOS X environment.

We can make another comparison between two different models of programming. In **structured programming**, blocks of programming statements (code) are executed one after another. Control statements (described later on) change which blocks of code are executed next.

In **object oriented programming**, data are contained in objects and are accessed using special methods (blocks of code) specific to the type of object. There is no single “flow” of the program as objects can freely interact with one another by passing messages.

In this tutorial, we focus only on structured programming.

Program Structure

Virtually all structured programs share a similar overall pattern:

- Statements to establish the start of the program
- Variable declaration
- Program statements (blocks of code)

The following is a simple example of a program written in several different programming languages. We call this the “Hello World” example since all the program does is print “Hello World” on the computer screen.

Language **Example program**

“C”

```
#include <stdio.h>
void main() {
    printf("Hello World");
}
```

C++

```
#include <iostream>
int main() {
    cout << "Hello World";
    return 0;
}
```

Pascal

```
program helloworld (output);
begin
    writeln('Hello World');
end.
```

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE helloworld AS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello World');
END;
```

Java

```
class helloworld {
    public static void main (String args []) {
       System.out.println ("Hello World");
    }
}
```

Perl

```
#!/usr/local/bin/perl -w print "Hello World";
```

Basic

```
print "Hello World"
```

Note that the Perl and Basic languages are technically not compiled languages. These language statements are “interpreted” as the program is running.

Variable Declaration

Variables are place holders for data a program might use or manipulate. Variables are given names so that we can assign values to them and refer to them later to read the values. Variables typically store values of a given *type*. Types generally include:

- Integer – to store integer or “whole” numbers
- Real – to store real or fractional numbers (also called float to indicate a floating point number)
- Character – A single character such as a letter of the alphabet or punctuation.
- String – A collection of characters

In order to use a variable within a program, the compiler needs to know in advance the type of data that will be stored in it. For this reason, we declare the variables at the start of the program. Variable declaration consists of giving a new name and a data type for the variable. This is normally done at the very start of the program.

In the following example programs, variables of different types are declared and used in the programs.

Language **Example program**

“C”

```
#include <stdio.h>
void main() {
   int age;
   float salary;
   char middle_initial;
   age = 21;
   salary = 29521.12;
   middle_initial = "K";
   printf("I am %d years old ", age);
   printf("I make %8.2f per year " salary);
   printf("My middle initial is %c ", middle_initial);
}
```

Pascal

```
program myexample (output);
   var age: integer;
   salary: real;
   middle_initial: char;
begin
   age := 21;
   salary := 29521.12;
   middle_initial := 'K';
   writeln('I am ', age, ' years old');
   writeln('I make ', salary, ' per year');
   writeln('My middle initial is ', middle_initial);
end.
```

Oracle PL/SQL

```
CREATE OR REPLACE PROCEDURE myexample AS
    age NUMBER;
    salary NUMBER;
    middle_initial CHAR(1);
BEGIN
   age := 21;
   salary := 29521.12;
   middle_initial := 'K';
   DBMS_OUTPUT.PUT_LINE('I am ' || age || ' years old');
   DBMS_OUTPUT.PUT_LINE('I make ' || TO_CHAR(salary, '99999.99') || ' per year');
   DBMS_OUTPUT.PUT_LINE('My middle initial is ' || middle_initial);
END;
```

Java

```
class myexample {
   public static void main (String args []) {
      int age;
      double salary;
      String middle_initial;
      age = 21;
      salary = 29521.12;
      middle_initial = "K";
      System.out.println ("I am " + age + " years old ");
      System.out.println ("I make " + salary + " per year");
      System.out.println ("My middle initial is " + middle_initial);
   }
}
```

Perl

```
#!/usr/local/bin/perl $
     age = 21;
     $salary = 29521.12;
     $middle_initial = "K";
     print "I am " . $age . " years old ";
     print "I make " . $salary . " per year ";
     print "My middle initial is " . $middle_initial;
```

Basic

```
Dim age AS Integer
Dim salary AS Decimal
Dim middle_initial As String
age = 21
salary = 29521.12
middle_initial = "K"
print "I am " & age & " years old "
print "I make " & $salary & " per year "
print "My middle initial is " & $middle_initial
```

In the above examples it is clear that different programming languages have slightly different *syntax* and data types. However for the most part, variable declaration is straight forward.

Real differences begin to appear when more complex data structures such as arrays and pointers are declared. Such discussion is best left for programming courses.

Variables are place holders for data a program might use or manipulate. Variables are given names so that we can assign values to them and refer to them later to read the values. Variables typically store values of a given *type*. Types generally include:

Boolean Logic and Boolean Algebra

Boolean Algebra was invented by nineteenth century mathematician George Boole. In Boolean Algebra, mathematical expressions are evaluated to one of two values: True or False. Boolean logic is the mathematical logic that is fundamental to Boolean Algebra.

Boolean logic is used throughout computer science. Understanding how to pose and evaluate Boolean logic expressions is a crucial skill for any programmer or indeed anyone who programs computers in a formal language (such as “C”, C++, Java, Pascal, Fortran, SQL, etc.) or in a macro language such as shell scripts or MS Excel formulas and macros.

This primer endeavors to introduce the topic of Boolean logic and demonstrates how it is used in a variety of situations.

Boolean logic is called a *Two Valued Logic* because an expression may only take on one of two values: True or False. An expression is some collection of logical operands and logical operators that are combined together.

In arithmetic, the operands are numbers and the operators are the familiar addition, subtraction, multiplication and division. In Boolean logic, the operands are statements (that can be proven True or False) and the operators are logical AND, OR and NOT.

For example consider this expression: `4 < 6`
clearly this expression is True since the number four is less than the number 6.

As another example, consider the following statement:
I am 6 feet tall AND I am president of the United States

While it may be True that I am six feet tall (a fact that can be proven by measuring my height), it can certainly be shown that I am not the president of the United States. Therefore, according to Boolean logic, this entire sentence is *False*. Another way of saying this is: It is False that I am 6 feet tall AND I am the president of the United States.

Consider a similar sentence:
I am 6 feet tall OR I am president of the United States

Notice in this case that only one of the parts of the sentence (separated by OR) need be True in order for the entire sentence to be considered True. Another way of saying this is: It is True that I am 6 feet tall OR I am the president of the United States.

The Boolean Operators

There are three Boolean operators: AND, OR and NOT. These operators are written differently depending on the language being used. In mathematics, the logical operators are written as $\wedge$ $\vee$ $\neg$

The following table compares how AND OR and NOT are written in different programming languages:

Language	AND	OR	NOT
Mathematics	$\wedge$	$\vee$	$\neg$
"C" or C++	&&	||	!
SQL	AND	OR	NOT
Pascal	AND	OR	NOT
Perl	&	||	!
Java	&	||	!
Basic	AND	OR	NOT

The Boolean operators are evaluated in the following fashion. For the AND operator, the combination of two "True" values results in "True" all other combinations evaluate to False. For example:
`True AND True` evaluates to True
`True AND False` evaluates to False
`False AND True` evaluates to False
`False AND False` evaluates to False

For the OR operator, as long as one of the value is True, then the expression evaluates to True:
`True OR True` evaluates to True
`True OR False` evaluates to True
`False OR True` evaluates to True
`False OR False` evaluates to False

The NOT operator is called the “complimentary” operator. It reverses the truth value of the operand:
`NOT True` evaluates to False
`NOT False` evaluates to True.

The Boolean operators and their evaluations are often expressed in Truth Tables. We write **T** and **F** to signify the values of True and False respectively. The following are truth tables for the three operators:

$\wedge$	**T**	**F**
T	T	F
F	F	F

$\vee$	**T**	**F**
T	T	T
F	T	F

$\neg$	**T**	**F**
	F	T

To use a truth table, choose a value from a row on the top and from a column on the left side and then see where they intersect to determine what they evaluate to. For example, looking at the truth table for AND, pick the value **F** for False on the top and then pick the value **T** for True along the left side. Where these two intersect shows an **F** for False meaning *True AND False evaluates to False*.

Comparison Operators

Boolean expressions often involve comparison operators that can be evaluated to determine if they are True or False. Comparison operators include: = < > ≥ ≤ ≠

Different programming languages write these comparison operators in different ways:

Language	Equality	Greater than	Less than	Greater than or equal to	Less than or equal to	Inequality
Mathematics	=	>	<	≥	≤	≠
"C" or C++	==	>	<	>=	<=	!=
Pascal	=	>	<	>=	<=	<>
SQL	=	>	<	>=	<=	<>
Perl	== or `eq`	> or `gt`	< or `lt`	>= or `ge`	<= or `le`	!= or `ne`
Java	==	>	<	>=	<=	!=
Basic	=	>	<	>=	<=	!=
Python	==	>	<	>=	<=	!=

Note for Perl language comparison of numbers uses the typical mathematical operators. Comparison of strings uses the `eq`, `ge`, `le`, `gt`, `lt` and `ne` operators.

Comparison operators are used to form expressions that can be evaluated as True or False. For example, we might ask if there are more than 30 students in the class using the following expression:

```
Number_of_Students > 30
```

If there are more than 30 students in the class, then this expression will evaluate to True. If there are 30 or fewer students in the class, then this expression will evaluate to False.

The following are more examples of these types of expressions:

Question	**Expression**
Does Alice make more than $35,000 per year?	`Alice_Salary > 35000`
Did the NY Giants defeat the Dallas Cowboys?	`Giants_Points > Cowboys_Points`
Did anyone get a perfect score on the test?	`TestScore = 100`

For each of these examples, supplying values for each of the expressions allows us to determine if that expression is True or False. For example, if Alice only makes $31,000, then the first expression would evaluate to False.

Combining Boolean and Comparison Operators

In the previous sections, we have seen how the Boolean operators and Comparison operators can be used to form expressions. In this section, we combine the two types of operators to form more complex expressions.

For example, suppose we ask the question: Are there more than 30 students in the class, and are there more than 30 seats in the room? This might be expressed as:

```
Number_of_Students > 30 AND Number_of_Seats > 30
```

We evaluate such an expression by determining the value (True or False) of each of the comparison expressions, and then use those values to evaluate the Boolean expression. In our example, suppose the class is a very large one that is held in a large room. In other words, there are more than 30 students and the room has more than 30 seats. Therefore, we would evaluate the expression as follows:

1. The `Number_of_Students > 30` portion of the expression evaluates to True.
2. The `Number_of_Seats > 30` portion of the expression evaluates to True.
3. Substituting those values in the expression give us:
   `True AND True`
4. The `True AND True` expression evaluates to True as can be seen in the Truth tables for the AND operator.
5. Therefore, we conclude that the entire expression evaluates to True.

The following example expressions are based on these assumptions:

1. All employees belong to a department
2. All employees working in department 5 have salaries greater than $25,000
3. All employees in department 4 make exactly $40,000 per year.
4. Alice is an employee and she works in department 5
5. Bill is an employee and he works in department 4

Given the above 5 items are facts, consider the truth values of the following expressions:

Expression	**Evaluation**
`Alice_salary < 25000 AND Alice_Department = 5`	`Alice_salary < 25000` is False `Alice_Department = 5` is True `False AND True` is False
`NOT Alice_salary < 25000`	`Alice_salary < 25000` is False `NOT False` is True
`Alice_salary < 25000 OR Bill_Salary = 40000`	`Alice_salary < 25000` is False `Bill_Salary = 40000` is True `False OR True` is True
`Alice_salary > 25000 AND NOT Bill_Salary < 40000`	`Alice_salary > 25000` is True `Bill_Salary < 40000` is False `NOT False` is True `True AND True` is True

As an exercise, evaluate the following expressions:

Expression	**Evaluation**

`Alice_salary < 25000 OR`
`Alice_Department = 4`	

`Alice_salary >= 25000 AND`
`Bill_department = 4`	

`Alice_salary < 25000 OR`
`Bill_Salary < 40000 OR`
`Alice_Department = 5`	

`NOT ( Alice_salary > 25000 AND`
`Bill_Salary < 40000 )`	

`Alice_salary < 25000 AND`
`( Bill_Salary <= 40000 OR`
`Alice_Department = 5 )`	

Notice in the last two cases, parenthesis may be used to group portions of the expression. In general Boolean expressions are evaluated from left to right except where parenthesis change the order.

In the third example above, notice that when multiple conditions are separated by OR, it only takes one of the conditions to be True in order to cause the entire expression to evaluate to True.

Conditional Statements (IF..THEN..ELSE)

In the prior sections, expressions combining Boolean operators and comparison operators were described. In this section, we demonstrate how such expressions are written and evaluated in computer programming languages.

Control statements "control" which sections of code in a program are executed. For example, we might want a set of statements to execute only if certain conditions are met, otherwise, the statements would not be executed. There are three general types of control statements:

1. Sequential – The default ordering of execution
2. Decision (Conditional) – controls which block of code within several alternatives is executed.
3. Iterative – controls how many times a block of code is executed.

Decision or Conditional statements are a type of Control statement and are often referred to as IF..THEN..ELSE statements as can be seen in these examples:

Programming Language — **Conditional Code**

"C", C++, Java, Perl

```
if (condition) {
   statements
} else {
    else_statements
}
```

Pascal

```
if (condition) then
begin
   statements
end;
```

Oracle PL/SQL

```
if (condition) then
   statements
else
   statements
end if;
```

Basic

```
If condition Then
   statements
Else
   statements
End If
```

In each of the languages, the *condition* portion can be an arbitrarily complex boolean expression (using variables declared in the program) that will evaluate to either True or False. If*condition* evaluates to True then the *statements* will be executed. If *condition* evaluates to False, then the *else_statements* will be executed if any are given.

In the following examples, a variable `department` is defined and a value is assigned to it. A conditional statement then determines which message should be displayed.

Language **Example program**

“C” or C++

```
#include <stdio.h>
main() {
      int department;
      department = 5;
      if (department == 5) {
         printf("Employees should be paid more than 25,000");
      } else {
         printf("Employees should be paid exactly 40,000");
      }
}
```

Perl

```
$department = 5;
if ($department == 5) {
   print "Employees should be paid more than 25,000";
} else {
   print "Employees should be paid exactly 40,000";
}
```

Oracle PL/SQL

```
BEGIN
   DECLARE
     department INTEGER;
     department = 5;
     IF (department = 5) THEN
        DBMS_OUTPUT.PUT_LINE('Employees should be paid more than 25,000');
     ELSE
        DBMS_OUTPUT.PUT_LINE('Employees should be paid exactly 40,000');
     END IF;
   END;
```

Java	`class myexample {`
	`   public static void main (String args []) {`
	`      int department = 5;`
	`      if (department == 5) {`
	`         System.out.println ("Employees should be paid more than 25,000");`
	`      } else {`
	`         System.out.println ("Employees should be paid exactly 40,000");`
	`      }`
	`   }`
	`}`

SQL	`SELECT salary FROM employees_table WHERE department = 5;`

As an exercise, determine what the output should be from the following Java programs:

Example **Java Program**

1

```
class myexample1 {
   public static void main (String args []) {
      int bill_department = 4;
      int bill_salary = 40000;
      int alice_department = 5;
      int alice_salary = 51000;
      if ( (bill_department == 4) & (alice_department == 5) ) {
          System.out.println ("This statement is True!");
      } else {
          System.out.println ("This statement is False!");
      }
   }
}
```

2

```
class myexample2 {
   public static void main (String args []) {
      int bill_department = 4;
      int bill_salary = 40000;
      int alice_department = 5;
      int alice_salary = 51000;
      if ( (bill_salary < 40000) || (alice_department >= 5) ) {
         System.out.println ("This statement is True!");
      } else {
         System.out.println ("This statement is False!");
      }
   }
}
```

3

```
class myexample3 {
     public static void main (String args []) {
        int bill_department = 4;
        int bill_salary = 40000;
        int alice_department = 5;
        int alice_salary = 51000;
        if ( (alice_department == 5) || (alice_salary < 25000) || (bill_salary >= 40000) ) {
           System.out.println ("This statement is True!");
        } else {
           System.out.println ("This statement is False!");
        }
   }
}
```

Iterative Constructs (Loops)

Virtually all programming languages have a facility to allow a section of code to be repeated (iterated or looped). There are several variations to iterative or looping constructs. For the most part, they fall into two categories: FOR loops and WHILE/DO loops.

FOR loops

Simple FOR loops iterate a specific number of times based on counting up (or down) on an integer variable. For example, if we want to do something 10 times, we can make a loop that counts up from 1 to 10 and put our work (what we want done) inside the loop.

Generally the three main parts that need to be written for a FOR loop are the initialization (the initial or starting value), the condition (the condition under which the loop will stop) and the increment (or decrement).
The other main part of the syntax is some indication of the start of the loop section and the end of the loop section. In many languages these are indicated by opening and closing curly brackets `{` and `}`.

Below are some examples for different languages:

Programming Language	FOR loop
"C", C++, Java, Perl	`for (initialize; condition; increment) {` `statements` `}`
Pascal	`for variable := low_integer to high_integer do` `begin` `statements` `end;`
Oracle PL/SQL	`FOR variable IN  low_integer .. high_integer LOOP` `statements` `END LOOP;`
Basic	`For variable =  low_integer To high_integer` `statements` `Next variable`

"C", C++, Java, Perl
```
for (initialize; condition; increment) {
   statements
}
```

Pascal
```
for variable := low_integer to high_integer do
begin
     statements
end;
```

Oracle PL/SQL
```
FOR variable IN  low_integer .. high_integer LOOP
   statements
END LOOP;
```

Basic
```
For variable =  low_integer To high_integer
   statements
Next variable
```

In each of the following examples, the code will output "Hello World" 10 times.

Programming Language **Example**

"C", C++

```
for (i = 1; i <= 10; i = i+1) {
     printf("Hello World\n");
}
```

Java

```
for (i = 1; i <= 10; i = i+1) {
   System.out.println ("Hello World");
}
```

Pascal

```
for i := 1 to 10 do
begin
   writeln('Hello World');
end;
```

Oracle PL/SQL

```
FOR i IN 1..10 LOOP
   DBMS_OUTPUT.PUT_LINE('Hello World');
END LOOP;
```

Basic

```
For i = 1 To 10
   Print "Hello World"
Next i
```

WHILE and DO loops

Like FOR loops, the WHILE and DO loops are also used to repeat a section of code for some number of times. However while FOR loops basically specify some count, WHILE and DO loops repeat while a *condition* is true. The *condition* is a boolean expression that we can evaluate as either "true" or "false". As soon as the *condition* evaluates to "false" the loop ends.

The main difference between a WHILE loop and a DO loop is where the *condition* is checked. WHILE loops check the *condition* first, and then run the statements if true. DO loops run the statements first, then check if the *condition* is true.

Below are the general syntax

Programming Language	WHILE loop	DO loop

"C", C++, Java, Perl	`while (condition) {`
	`    statements`
	`}`
		`do {`
		`    statements`
		`} while (condition)`

Pascal	`while (condition) do`
	`begin`
	` statements`
	`end;`
		`Repeat`
		` statements`
		`until  (condition);`

Oracle PL/SQL	`WHILE condition LOOP`
	`    statements`
	`END LOOP;`
		`LOOP`
		`    statements`
		`    EXIT WHEN  condition ;`
		`END LOOP;`

Basic	`While condition`
	`    statements`
	`Wend`
		`Do`
		`    statements`
		`Loop While condition`

WHILE loops can be set up to emulate what a FOR loop is doing. For example, the following "C" or C++ code prints "Hello World" 10 times:

Programming Language	FOR Loop	WHILE Loop

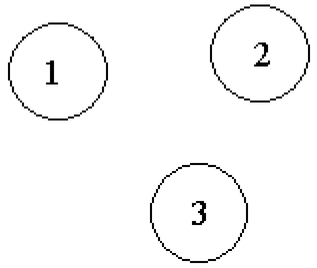
"C", C++	`for (i = 1; i <= 10; i = i+1) {`
	`     printf("Hello World\n");`
	`}`
		`i = 1;`
		`while (i <= 10) {`
		`     printf("Hello World\n");`
		`     i = i + 1;`
		`}`

Graph Theory Basics

Many aspects of computer science, and especially programming can be explained or demonstrated with the aid of graph theory. What follows is a very brief "layman's introduction" to graph theory.

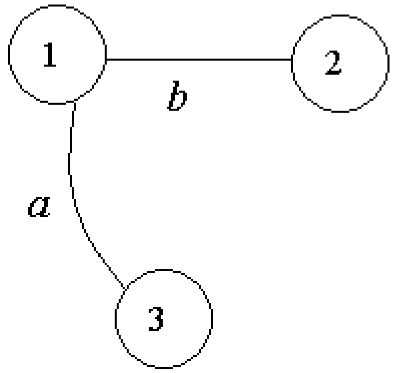
Nodes and Arcs

Graph theory is based on the notion of *Nodes* (vertices) and *arcs* (edges). A node represents some *state* of the system. It could represent the value of a variable or a particular situation. We generally draw a node as a circle with some label inside of it. In the following figure, three nodes (labeled, *1*, *2* and *3*) are defined.



Arcs represent a way to transition from one node to the next. They can represent actions that move form one node to another or relationships between nodes. Arcs always begin and end at some node. It's possible for an arc to begin and end at the same node.

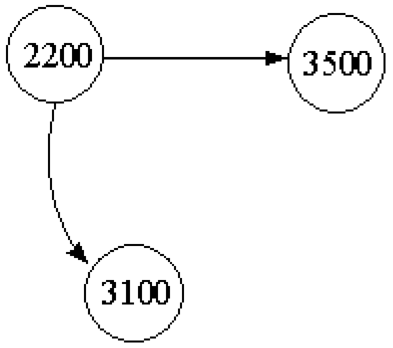
In the following figure, nodes 1, 2 and 3 of the graph have been joined by arcs labeled *a* and *b*.



An example explanation might be that each node represents a person where nodes 1 and 2 “know” each other and nodes 1 and 3 “know” each other. However, nodes 2 and 3 have not yet been introduced.

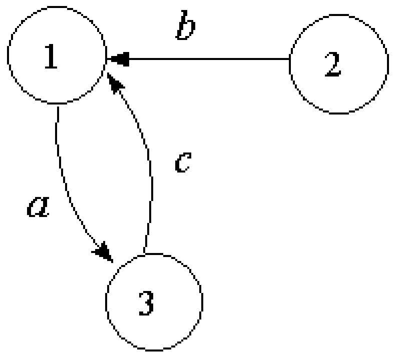
Arc can also have a direction to them. This is indicated by an arrow head on one or both of the ends of the arc. A graph with directions to the arcs is called a *directed graph*. Directed graphs can be used to show a precedence relationship (e.g., which comes first?) or causal relationships (which node “causes” which other node).

In the following figure, each node represents a CIS course at Baruch College. The direction of the arrows indicates which courses are pre-requisites.



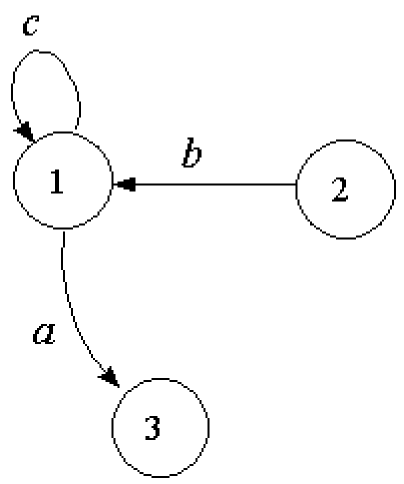
In this example, it is clear that a student must take (and pass!) CIS 2200 before he or she can take either CIS 3100 or CIS 3500.

A *cycle* occurs when one can start at a node and follow a *path* of arcs that lead back to the starting node. A cycle is sometimes called a *circuit*. We can write a cycle in terms of the nodes that are visited. For example, the following graph shows the cycle: `1 -> 3 -> 1`



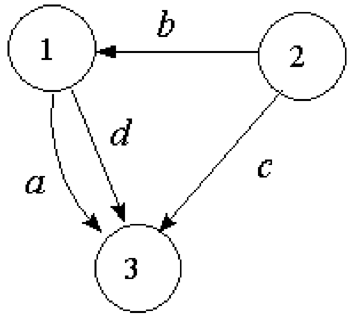
That is, by starting at node 1 and following arc *a*, then following arc *c*, we wind up back at node 1 again. Therefore, we say that this graph has a cycle.

It is possible for an arc to start and end at the same node as in the following example:



Here arc *c* is a directed arc that begins and ends at node 1. This graph also has a cycle: `1 -> 1`

A graph that does not contain any cycles is called *acyclic*. The following is an example of an acyclic graph:



From the perspective of a node, we can view an arc as either *incoming* or *outgoing*. An incoming arc is pointing to the node. An outgoing arc is pointing away from the node. In the above figure: Node 1 has two outgoing arcs (*a* and *d*), and one incoming arc (*b*). Node 2 has two outgoing arcs (*b* and *c*), but no incoming arcs. Node 3 has no outgoing arcs and three incoming arcs (*a*, *c* and *d*).

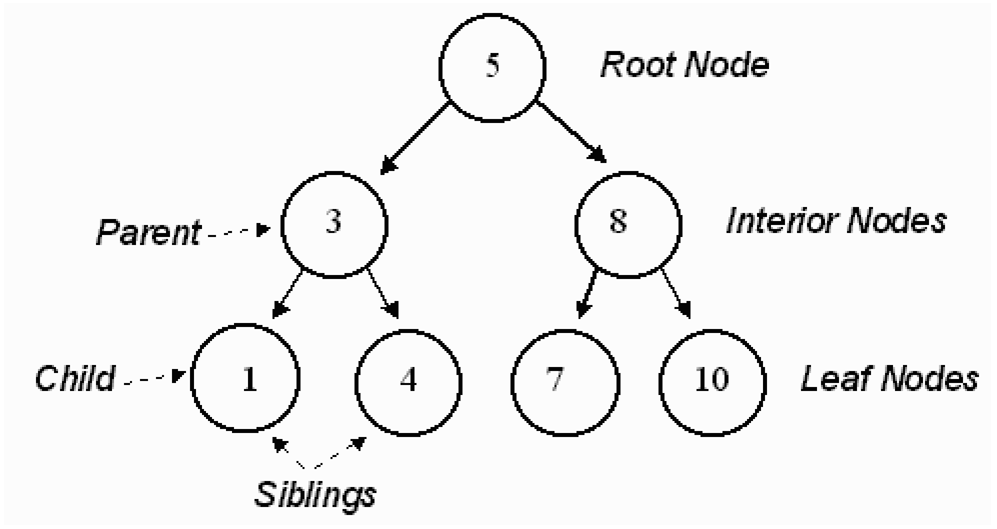
A node with no incoming arcs but one or more outgoing arcs is called a *source*. A node with no outgoing arcs but one or more incoming arcs is called a *sink*.

In the above diagram, node 2 is a source and node 3 is a sink.

Tree Structures

Tree structures are quite common in computer science. They can represent a sorted collection of numbers or names, or a series of decisions that need to be made. Like graphs (described in another section), trees are made up of nodes and directed arcs. Nodes represent some value and arcs point from one node to another. Trees are in many ways like graphs with restrictions on how the arcs are arranged.

In computer science trees grow upside down. The root is at the top and the leaves are at the bottom. Below is an example of a tree that is being used to hold some integers in a particular order:



The nodes in this tree represent integers. In this case, the integers 1, 3, 4, 5, 7, 8 and 10 have been used. The *root node* is labeled as “5” and appears at the top of the tree. Several *interior nodes* (nodes that are neither root nor leaves) labeled 3 and 8 appear in the middle of the tree. Finally, *leaf nodes* with labels 1, 4, 7 and 10 appear at the bottom of the tree.

A *Parent node* is one that has at least one node below it. In the above tree, node 3 is a parent and its *Child nodes* are 1 and 4. (Can you find the other 2 parent nodes in the graph?) Child nodes with a common parent are called *sibling nodes*. In the example, nodes 1 and 4 are siblings. As a rule, child nodes in a tree may only have one parent. (Can you find the other siblings in the graph?)

Another feature of a tree is the number of levels of depth. In the example, the tree is 3 levels deep. If the number of children is the same for all parent nodes, we say that the tree is *balanced*. In the above example, all nodes (except the leaf nodes) have 2 children so the tree is balanced.

A tree with 2 children per parent is also called a *binary* tree. Looking at the above example, one can see that for each parent node, one of its children has a value less than the parent, and the other child has a value greater than that of the parent. For example, the parent node 5 has a *left child* with value 3 (less than its parent) and a *right child* with value 8 (greater than its parent).

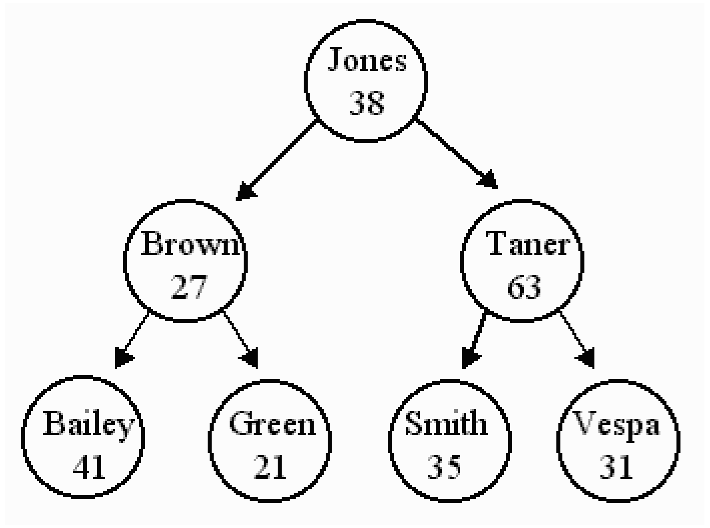
Trees can be used to assist us in sorting and searching for data of interest. For example, suppose we have the following set of data:

Name	**Age**
Bailey	41
Brown	27
Green	21
Jones	38
Smith	35
Taner	63
Vespa	31

We would like to be able to store this data in such a way that improves how we can search for a person later on. We can construct a binary tree to store the names and ages in the following fashion. First, find the median value (the value that appears closest to the middle) of the list and make that the root node. In this example, "Jones" appears to be in the middle of the list so we can make Jones the root node.

Next, look at the all of the values that come before Jones in the list: Bailey, Brown and Green. Choose the median of these three and make it the left child of Jones. Repeat this process with the bottom half of the list (after Jones) to establish the right child of Jones.

Finally, repeat the process again for the next level by taking Brown and Taner each as parents. The resulting tree looks like the following:



To search such a tree, start at the root node and:

1. Compare the target value (what is being searched for) with the root node.
2. If the target value matches the current node, the search is over
3. If the target value is less than the current node's value, move to the current node's left child and go back to step 2.
4. If the target value is greater than the current node's value, move to the current node's right child and go back to step 2.
5. If we reach a leaf node and the target value is not found, then it is not in the tree.

As an example, assume we are searching for "Vespa". Starting with the root node, we compare Vespa with Jones. Since Vespa comes after Jones in the alphabet (Vespa is greater than Jones) we move to Jones' right child. At this node, we compare Vespa with Taner and see that since Vespa is greater than Taner, we move to Taner's right child node. Finally, we find the target node.

Notice that if we have to search through the list (starting from the first entry), it would have taken us 7 steps to finally locate Vespa. On average, we would expect to find what we are looking for in the list after about 3 or 4 steps. With the binary tree, we will never have to use more than 3 steps.

We can generalize this reasoning with the following points:

- Searching in a list (linear search) of *n* items will require on average *n / 2* steps.
- Searching the same list represented as a binary tree will require no more than $\log_2 n$ steps.

Here is a comparison:

Number of Items *n*	**Linear Search**	**Binary Search**
4	4/2 = 2	$\log_2 4 = 2$
8	8/2 = 4	$\log_2 8 = 3$
16	16/2 = 8	$\log_2 16 = 4$
32	32/2 = 16	$\log_2 32 = 5$
64	64/2 = 32	$\log_2 64 = 6$
128	128/2 = 64	$\log_2 128 = 7$

It is clear from the comparison that the binary tree representation can cut down significantly on the number of steps.

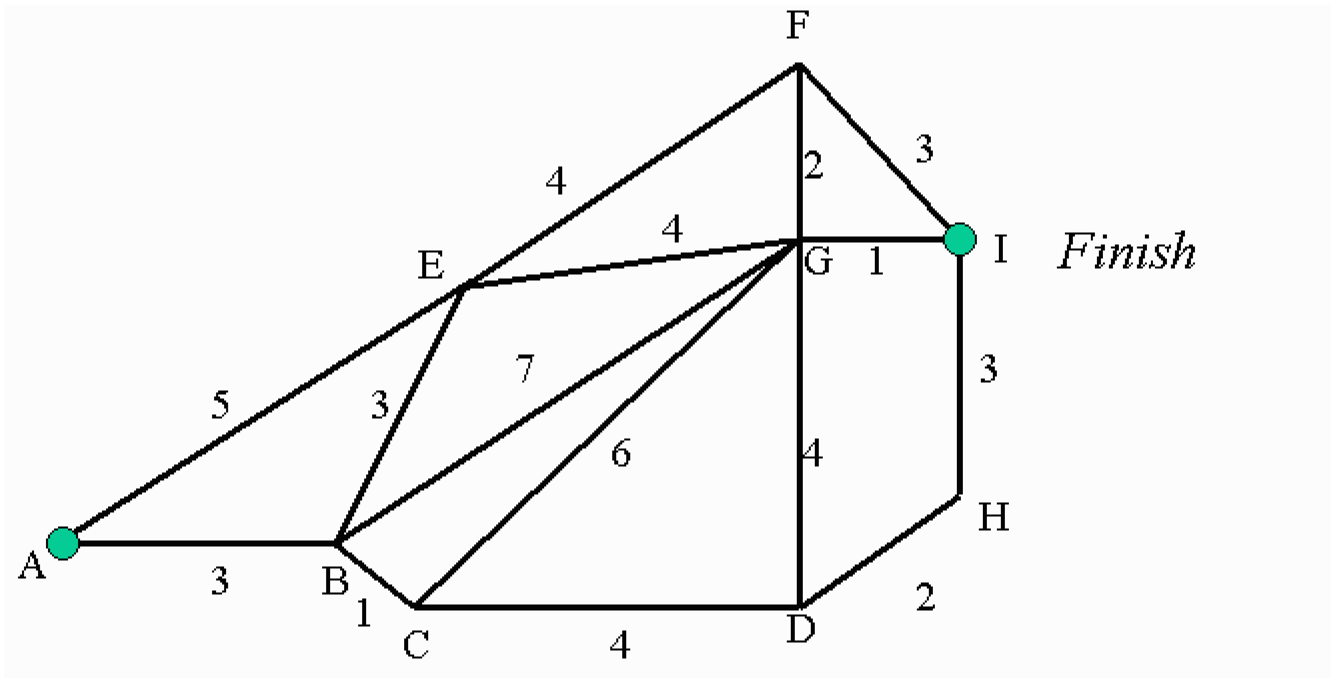
Optimization

Optimization refers to finding the best solution to a problem given a set of possible alternatives. There are many examples of optimization problems such as transportation problems, routing network packets, investing capital, allocating scarce resources and many others. The example presented below is a type of transportation optimization problem where the variables used are integers. These types of optimization problems are called combinatorial.

An Optimization Example

The purpose of this example is to give a basic idea of a problem in optimization and to illustrate the concepts of *solution space*, and *brute force* and *heuristic* approaches to solving optimization problems.

The graph below consists of 9 nodes labeled A, B, C, D, E, F, G, H and I and 15 edges. Each of the edges has some weight associated with it. For example, the edge between node A and node E has weight of 5. The edge between node G and node I has a weight of 1. We might consider each weight as some kind of cost such as the number of miles a car must drive.



The goal of this optimization problem is to identify a path starting from node A and finishing at node I without crossing the same edge twice. As the path is traversed, we sum up the weights along the way. The optimal solution to this problem will be the path with the least total cost.

Notice that for this problem, there are many possible solutions:

A -> E -> F -> I Cost: 5 + 4 + 3 = 12
A -> E -> G -> I Cost: 5 + 4 + 1 = 10
A -> B -> G -> I Cost: 3 + 7 + 1 = 11
A -> B -> C -> D -> G -> I Cost: 3 + 1 + 4 + 4 + 1 = 13
A -> B -> C -> D -> H -> I Cost: 3 + 1 + 4 + 2 + 3 = 13
A -> E -> F -> G -> B -> C -> D -> H -> I Cost: 5 + 4 + 2 + 7 + 1 + 4 + 2 + 3 = 28 and so on…

Even with just nine nodes as in this example, a large number of solutions are possible. The number of potential solutions grows very rapidly as the number of nodes increases. We call this set of possible solutions the *solution space*. The goal is to choose the least cost solution from this solution space.

There are two main approaches to finding the least cost path. We might consider a *brute force* method where every path is enumerated and the cost is compared with others to see if it is the least cost. Such a method is guaranteed to identify the absolute lowest cost path. However, if the solution space is large, it could take us an extremely long time to identify this lowest cost path.

Another approach to solving the problem is to use a *heuristic* or a set of general rules that can guide us to a solution. For example, one heuristic would be at any given node, choose the path with the least cost. in the example graph above, such a heuristic results in a solution:
`A -> B -> C -> D -> H -> I` with a total Cost:
`3 + 1 + 4 + 2 + 3 = 13`

While this solution is not the optimal solution, it can be reached in a very short time as compared with the brute force approach. It is possible that a heuristic approach can each a solution that is much worse than the optimal solution. However, faced with the alternative of spending significant time locating the optimal solution, we may be willing to accept a sub-optimal solution if it can be reached in a short time.

One application of this heuristic approach is in the optimization of database queries. Given a database query, the database management system may have a number of ways in which to carry out the work required to achieve the query result. A heuristic approach may look at some simple statistics such as the number of records in a table and optimize the query based on working on the table with the smallest number of records first. A brute force approach would be to explore the estimated costs of performing the query steps in different orders and then selecting the ordering of the steps that minimizes the total amount of work.

The Binary Number System

Computers store and operate on data represented in the binary number system. A great many aspects of computer science require some understanding of the binary number system. In order to understand the binary number system, it is often helpful to examine a number system we are all familiar with.

At some point in grammar school, you learned the “decimal” number system. Such a number system is based upon units of “10”. In mathematics and computer science we call the decimal number system the “Base 10” number system. We use the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 to represent numbers in the following way:

Digits	**Representation**
0	Value of 0
1	Value of 1
2	Value of 2
3	Value of 3
4	Value of 4
5	Value of 5
6	Value of 6
7	Value of 7
8	Value of 8
9	Value of 9
10	One “group” of 10 followed by the value of 0
11	One “group” of 10 followed by the value of 1
…	…
20	Two “groups” of 10 followed by the value of 0

…	…
100	One “group” of 100 followed by zero groups of 10 followed by the value of 0
…	…
125	One “group” of 100 followed by 2 groups of 10 followed by the value of 5

In other words, when we work with the decimal (Base 10) number system, we tend to think in groups of 10, 100, 1000 and so on. We count up using the symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Each successive “place” in a number is a multiple of 10. That is, we have the “ones”. “tens”, “hundreds”, “thousands” and so on.

```
Number:  125

Hundreds       Tens       Ones
  1              2          5

Number:  3461

Thousands    Hundreds        Tens        Ones
    3            4             6           1
```

In the binary (Base 2) number system, a digit can only take on one of two values: Either 0 or 1. We call such a digit a “bit” (short for “Binary Digit”). Each successive “bit” in a binary number is a multiple of 2. That is we have the “ones”, “twos”, “fours”, “eights”, “sixteens”, “thirtytwos” and so on.

Below are some examples of binary numbers:

Decimal Digits	**Binary Digits**	**Representation**
0	0	Value of 0
1	1	Value of 1
2	10	One “group” of 2 followed by the value 0
3	11	One “group” of 2 followed by the value 1
4	100	One “group” of 4 followed by 0 groups of 2 followed by the value 0
5	101	One “group” of 4 followed by 0 groups of 2 followed by the value 1
6	110	One “group” of 4 followed by 1 group of 2 followed by the value 0
7	111	One “group” of 4 followed by 1 group of 2 followed by the value 1
8	1000	One “group” of 8, 0 groups of 4, 0 groups of 2 and the value 0
…	…	…
20	10100	One “group” of 16, 0 groups of 8, 1 group of 4, 0 groups of 2 and the value 0
…	…	…
100	1100100	1 group of 64, 1 group of 32, 0 groups of 16, 0 groups of 8, 1 group of 4, 0 groups of 2 and 0
…	…	…
125	1111101	1 group of 64, 1 group of 32, 1 group of 16, 1 group of 8, 1 group of 4, 0 groups of 2 and 1

Computer scientists routinely memorize powers of 2. The process is simply to start with 0, 1 and then to double the number:

```
0, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048,
4098, 8192, 16384, 32768, 65536 and so on...
```

You may have notices that the RAM memory in your PC is also quite likely a power of 2:

```
8   Megabytes of RAM
16  Megabytes of RAM
32  Megabytes of RAM
64  Megabytes of RAM
128 Megabytes of RAM
256 Megabytes of RAM
and so on
```

Some questions we commonly ask are:

- If I need to store the number 1 through 100, how many bits will I need?
  We see that with 7 bits, we can represent numbers from 0 to 127. So 7 bits is all we would need to store numbers from 1 to 100.
- What is the largest number I can represent using 16 bits?
  The fastest way to calculate this is to take 2 to the 16th power ($_2$16) and then subtract 1.
  $_2$ 16 = 65,536
  So the largest number we can represent with 16 bits is: 65,536 – 1 = 65,535
- In my database, I can store 28 records in one disk block (a fixed sized storage area on the disk). How many bits would be required to index all of the records in one block?
  One way to answer this is to just run up the powers of 2:
  0, 1, 2, 4, 8, 16, 32, 64…
  Since with 5 bits we can represent numbers from 0 to 31, (recall that 2 to the 5th power is 32), all we should need are 5 bits to index the 28 record entries in the block.
- In networking, version 4 of the TCP/IP protocol designates IP addresses that are 32 bits long. How many hosts can be defined using 32 bits?
- Also in networking, the ethernet LAN protocol specifies that every ethernet device have a 48-bit Media Access Control (MAC) address.

The Modulus Operator

Many programming projects that require manipulation of numbers wind up requiring the remainder of an integer division operator. The *modulus* operator fills this requirement in a programming expression. You learned about the modulus operator way back in grade school when you were first learning arithmetic and division. A problem such as “What is 7 divided by 2?” would at that time have resulted in an answer of “3 with 1 left over” or “3 with remainder of 1”. It is the part that is “left over”, the remainder that the modulus operator returns.

The modulus operator is written differently in different programming languages, although most often the ampersand sign is used. Here are some examples:

Language	**Modulus Operator**	**Example expression**
“C” or C++	%	remainder = quotient % divisor ;

SQL	%	SELECT quotient % divisor FROM …
		Some DBMS such as Oracle implement it as a function as in:
		SELECT MOD(quotient, divisor) FROM …

Pascal	MOD	remainder := quotient MOD divisor ;
Perl	%	$remainder = $quotient % $divisor ;
Java	%	remainder = quotient % divisor ;

Bitwise Operators and bit manipulation

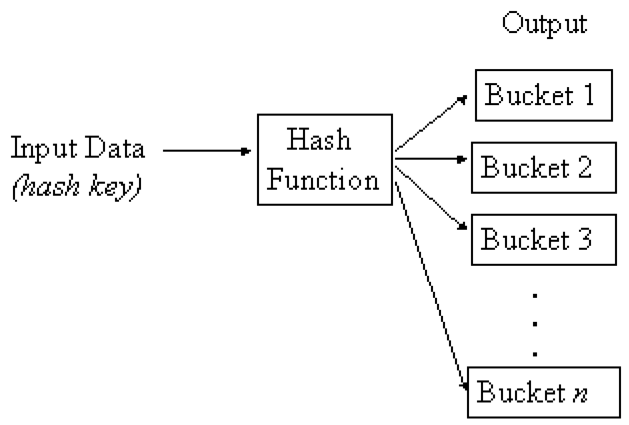
At their lowest level, computers manipulate bits – values that are either 1 or 0. There are many programming situations where manipulating individual bits becomes important. For example, in an industrial control system, individual bits might represent the state of a machine or the value of a particular sensor. A home alarm system might represent each door and window sensor with a bit where “0” represents a closed door or window and “1” represents an open door or window. In relational databases used for data warehousing, a set of bits are used to represent identifiers for data records in what are called “Bitmap indexes”.

Virtually every programming language provides operators to manipulate individual bits. The basic Boolean operators AND, OR and NOT were introduced in the prior section on Boolean Algebra. The same operators can be used to isolate and manipulate individual bits.

Hashing Functions

A hashing function maps some input data to a (smaller) output data set.

A very simple hashing function takes the input data and does a modulous operator on it.



Consider for example, office numbers and a simple hash function:

Input (office number)	**Hash function**	**Output**
245	245 mod 10	5
248	248 mod 10	8
253	253 mod 10	3

When two input values hash to the same output value, we call it a “collision”. Good hashing functions minimize collisions as well as the overall number of buckets.

In databases, hash functions are useful to quickly look up data records given an input key value.

In cryptography, hash functions are used to create message digests, digital signatures, responses to authentication challenges, etc.

To see a hashing function in action, under Linux:

```
[vps@server]$   echo "input" | md5sum
ec9187a89c2f150e910b6db5f37521b9

[vps@server]$   echo "inpuT" | md5sum
ce859c67402fbf6cafd57d55933e8aba
```

The above implements the MD5 hashing algorithm. Note that even a small change in input results in a drastic change in the output of the MD5 hash.