COMPUTER SCIENCE

(ITC315118)

Information Booklet

Last updated: 13/7/18

TABLE OF CONTENTS

ALGORITHMS

An algorithm is a set of steps to accomplish a task. Algorithms can help us to break down a problem and evaluate the effectiveness of a particular approach.

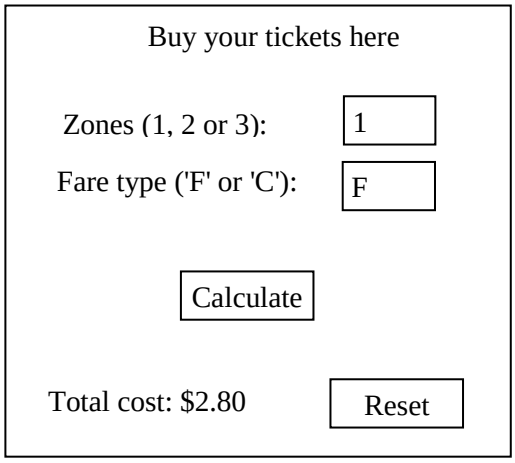
The *initially/when* model

In an *event-driven* program, the order in which actions are taken is determined by responses from the user when interacting with the program's GUI (Graphical User Interface). Event-driven programs should be described in terms of the *initial* state of the program and what happens *when* a particular condition occurs.

Algorithms for applets based on the *initially/when* model should work correctly no matter in what order the buttons or textfields are used.

Example - Bus tickets fare calculator

The following algorithm calculates the cost of a bus ticket based on the following specifications, assuming that valid data in entered. A possible screen for this applet is shown to the right.



Tickets can be purchased for 1, 2, 3 zones. The costs are:

- 1 zone => $2.80
- 2 zones => $3.50
- 3 zones => $4.20

Fares can be Full (i.e. full price) or Concession (50% discount).

```
Initially
    zones = 1
    fare_type = F
    cost = 2.80
    display cost

When a value is entered into the "zones" textfield
    set zones to value in "zones" textfield

When a value is entered into the "fare_type" textfield
    set fare_type to value in "fare_type" textfield

When the "calculate" button is pressed
    switch zones
        case '1': cost = 2.80
        case '2': cost = 3.50
        case '3': cost = 4.20
        default: display error
    if fare_type = 'C'
        cost = cost * 0.5
    display cost

When the "reset" button is pressed
    zones = 1
    fare_type = F
    cost = 2.80
    display cost
```

JAVA

Java is a high-level *object-oriented programming language*. It includes a set of class libraries that provide basic data types, system input and output capabilities, and other utility functions.

Java *applets* are programs that can be embedded in a web page, whereas an *application* is a standalone Java program. Both require a *Java virtual machine* (JVM) to run.

Java was intended to be *platform independent*. Java programs are compiled into *bytecodes* that can run on any operating system that has a *Java virtual machine* (JVM) installed, without needing recompilation. This is in contrast to platform dependent programming languages (e.g. C, C++) in which the source code is directly compiled into machine code.

Whereas Java is platform independent, the JVM is necessarily platform dependant.

Applets typically contain the following standard methods:

- The *init()* method is called once when the applet is created and is typically used to set up the user interface and define the initial values of any variables.
- The *paint()* method is used to display the applet. It runs automatically after the init() method and whenever the applet window is refreshed. It takes a Graphics object as parameter that can be used to draw shapes and text in the applet window. To update the display, the *repaint()* method can be used to force the paint() method to be called again.

Example - HelloWorld Applet

```
import java.awt.*;
import java.applet.*;

public class HelloWorld extends Applet
{
   public void paint(Graphics g)
   {
      g.drawString("Hello World", 20, 20);
   }
}
```

Keywords

abstract	const	finally	int	public	throw
boolean	continue	float	interface	return	throws
break	default	for	long	short	transient
byte	do	goto	native	static	true
byvalue	double	if	new	super	try
case	else	implements	null	switch	void
catch	extends	import	package	synchronized	volatile
char	false	instanceof	private	this	while
class	final		protected	threadsafe	

Comments

Java supports two types of comments:

```
/* This is a multiline
   comment.
*/
```

Everything between /* and */ is ignored by the compiler.

```
// single line comment
```

Everything following // to the end of the line is ignored by the compiler.

Variables and data types

A *variable* is a location of memory that is associated with a name (or *identifier*) and contains some information (or *value*).

In Java, a variable holds a value that is either a *primitive data type* or a *reference type*.

Java has eight primitive data types:

Type	Size	Range	Default value
boolean	1 bit	true or false	false
char	16-bit Unicode characters	Alphanumeric characters e.g. the characters on the keyboard	0 ("\u0000")
byte	8-bit two's complement integer	-128 to 127	0
short	16-bit two's complement integer	-32768 to 32767	0
int	32-bit two's complement integer	-2,147,483,648 to +2,147,483,647	0
long	64-bit two's complement integer	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	0
float	32-bit single precision floating point number (using IEEE 754 standard)	+/ - about 10^{39}	0.0
double	64-bit double precision floating point number (using IEEE 754 standard)	+/- about 10^{317}	0.0

Reference types include arrays and class instances (i.e. *objects*). Whereas a primitive variable holds exactly one value, a reference variable holds a *pointer* (or *reference*) to a location in memory that many contain multiple values and/or methods.

Variable names in Java:

- can contain letters, numbers, underscore (_), or the dollar sign ($)
- may only begin with a letter, an underscore or a dollar sign (by convention, variable names begin with a lowercase letter)
- are case-sensitive
- cannot be keywords.

The keyword *final* can used when declaring a variable to stop the value from being accidentally changed. By convention, the names of constants are typically written in uppercase.

```
final double PI = 3.14159;
double r = 20;
double areaOfCircle = PI * r * r;
```

Operators

This table shows the precedence for frequently-used Java operators. Operators higher up the table are evaluated before operators listed below them. Within an expression, operators with the same precedence are evaluated based on their associativity.

Operator	Description	Associativity

`.`
`[]`
`( )`
	member selection
	subscript
	method
		left-to-right

`++`
`--`
`!`
	increment
	decrement
	logical negation
		right-to-left

`(<type>)` `new`	typecasting object creation	

`*`
`/`
`%`
	multiplication
	division
	modulo (remainder)
		left-to-right

`+` `-`	addition or string concatenation subtraction	

`<`
`>`
`<=`
`>=`
	less than
	greater than
	less than or equal to
	greater than or equal to	

`==` `!=`	is equal to (equality) is not equal to (inequality)	
`&&`	logical AND	
`||`	logical OR	
`? :`	ternary (if-then-else)	right-to-left

`=`
`+=`
`-=`
`*=`
`/=`
`%=`
	assignment
	addition assignment (x = x + y)
	subtraction assignment
	multiplication assignment
	division assignment
	modulus assignment	

Promoting and casting

When expressions that contain different primitive data types are evaluated by the compiler, each value in the expression is automatically *promoted* (converted) to the larger type involved in the expression where possible. This is called a *widening conversion* and has no permanent effect on the variables in the expression. A widening conversion changes a value to a data type that will not lose information about the magnitude of the value (e.g. *int* to *double*).

Java's widening conversions are:

- byte to short, int, long, float, or double
- short to int, long, float, or double
- char to int, long, float, or double
- int to long, float, or double
- long to float or double
- float to double

Converting a larger primitive type to a smaller primitive type is called a *narrowing conversion*. This type of conversion may lose information, because it involves changing a value to a data type that might not be able to hold some of the possible values. To perform a narrowing conversion (e.g. *double* to *int*), a *cast operator*

is required. A cast operator consists of the name of the smaller type in brackets, placed directly before the value to be converted.

```
double a = 2 * 2.4;           // a = 4.8
int b = (int)(2 * 2.4);       // b = 4
double c = 5 / 4;             // integer division, c = 1.0
double d = 5.0 / 4.0;         // floating-point division, d = 1.25
int e = 5 % 4;                // modulo, e = 1
```

Another use of casting is to convert a character to its ASCII/Unicode value or the reverse.

```
char charValue = (char) 65;   // charValue = 'A'
int numValue = (int) 'B';     // numValue = 66
```

Any type may be converted to type String by *string conversion*. If only one value in an expression is of type String, then string conversion is performed on the other operand to produce a string at run time.

```
String s1 = "" + 42;          // s1 = "42"
String s2 = "" + 10 + 66;     // s2 = "1066"
String s2 = "" + (10 + 66);   // s2 = "76"
```

Arrays

An *array* is an object that contains a fixed number of values of a single data type.

The *index* of the first element in an array is always 0. If the length of the array is *n*, then the index of the last element is *(n – 1)*.

Example – Array of integers

```
int[] a = {6, 4, 9, 1, 5};

sum = 0;
for (int i = 0; i < a.length; i++)
    sum = sum + a[i];
```

Example – Two-dimensional array of integers

A two-dimensional array is effectively an array of arrays.

```
int[][] scores = {{93,74,77,55,81},
                  {78,77,72,75,80},
                  {92,88,82,83,69}}
int sum = 0;
for (int i=0; i<scores.length; i++)
    for (int j=0; j<scores[0].length; j++)
    {
        sum = sum + scores[i][j];
    }
}
```

Conditionals and loops

To write useful programs we almost always need ways to check conditions and change the behaviour of the program accordingly. We also frequently need to be able repeat sections of code.

The *if-else* statement

```
if (hours > 40)
   overtime = hours - 40;           // if with single statement

if (hours > 40)                     // if...else with multiple statements
{
   overtime = hours - 40;
   normal = 40;
}
else
{
   overtime = 0;
   normal = hours;
}
```

The *switch* statement

- Only works with some primitive data types (*byte*, *short*, *int*, and *char*) and the classes that wrap them (Byte, Short, Integer, Character), *enumerated types*, and the String class.
- The *break* statement terminates the execution of a switch statement. Without it, statements after the matching case are executed.
- Can have a default case that handles all other values.

```
switch (operation) {
    case '+':  g.drawString("5+6 = " + (5+6), 10, 10);
               break;
    case '-':  g.drawString("5-6 = " + (5-6),10,10);
               break;
    default:   g.drawString("wrong operator", 10,10);
               break;
}
```

The *for* statement

For statements define a loop that will run a predetermined number of times and describe the behaviour of the iterator in the loop header.

```
for (int i=1; i<=10; i++)
    g.drawString("Computing is great", 20, 20*i);
```

The *while* statement

In a *while* statement, the test condition is evaluated before entering the loop. If the condition is false when the loop is first entered it will not be executed.

```
while (condition) {
    statements;
}
```

The *do-while* statement

In a *do-while* statement, the test condition is evaluated after body of loop is executed so it is always performed at least once.

```
do {
    statements;
}
while (condition);
```

Strings

A *string* is a sequence of characters. Strings are *objects*, not one of the primitive data types.

Strings are *immutable*, which means that, once created, their values cannot be changed. String methods that appear to modify a string create and return a new string that contains the result of the operation.

Care needs to be taken when comparing strings. Because strings are objects, the comparison operator (==) only indicates whether two strings are the *same object*, not whether the character strings they contain are equal. To compare strings, instead use a String method, such as *equals()*, *equalsIgnoreCase()*, or *compareTo()*.

Strings can be *concatenated* (i.e. joined together) using the + symbol.

Selected String methods

Type	Method	Description
`char`	`charAt(int index)`	Gives the character at position number *index* in a string.
`int`	`compareTo(String string2)`	Returns 0 if the strings are alphabetically equal, a negative value if this string comes before *string2*, or a positive value otherwise.
`boolean`	`endsWith(String suffix)`	Tests if this string ends with the specified *suffix*.
`boolean`	`equals(String string2)`	Compares this string to *string2*.
`boolean`	`equalsIgnoreCase(String string2)`	Compares this string to *string2*, ignoring case.
`int`	`indexOf(String substring)`	Returns the index within this string of the first occurrence of the specified substring. Returns -1 if the *substring* isn't found.
`int`	`indexOf(String substring, int fromIndex)`	Returns the index within this string of the first occurrence of the specified character, starting the search at *fromIndex*. Returns -1 if the *substring* isn't found.
`boolean`	`isEmpty()`	Returns true if, and only if, length() is 0.
`int`	`length()`	Returns the length of this string.
`String`	`replace(char old, char change)`	Returns a new string resulting from replacing all occurrences of *old* in this string with *change*.
`boolean`	`startsWith(String prefix)`	Tests if this string ends with the specified *prefix*.
`String`	`substring(int beginIndex)`	Returns a substring of this string, starting at *beginIndex*.
`String`	`substring(int beginIndex, int endIndex)`	Returns a substring of this string, starting at *beginIndex*, and finishing at *endIndex*-1.
`char[]`	`toCharArray()`	Converts this string to a new character array.
`String`	`toUpperCase()`	Returns a copy of this string converted to upper case.
`String`	`toLowerCase()`	Returns a copy of this string converted to lower case.
`String`	`trim()`	Returns a copy of this string with all leading and trailing whitespace removed.

String examples

```
String s1 = "Everybody! ";
String s;
char   c;
int    n;
boolean b;

n = s1.length;                      // n = 11
s = s1.toUpperCase();               // s = "EVERYBODY! "
s = s1.replace('e', 'a');           // s = "Evarybody! "
s = s1.trim();                      // s = "Everybody!"
c = s1.charAt(5);                   // c = 'b'
s = s1.substring(5);                // s = "body! "
s = s1.substring(0,5);              // s = "Every"
b = s1.startsWith("Every");         // b = true
b = s1.endsWith("body!");           // b = false
b = s1.trim().endsWith("body!");    // b = true
n = s1.indexOf("very");             // n = 1
n = s1.indexOf("very",4);           // n = -1
```

String conversion

Objects of type String contain an array of char values. Strings can be converted to and from a character array.

```
char[] chars1 = {'o','b','j','e','c','t'};
String s2 = new String(chars1);                 // s2 = "object"
String s3 = new String(chars1,1,4);             // s3 = "bjec"
char[] chars2 = s3.toCharArray();               // chars2 = {'b','j','e','c'}
```

For each of the primitive data types (e.g. int, float, double) there is a corresponding wrapper class (e.g. Integer, Float, Double) that includes methods that may be used to convert between to and from strings.

```
int    n1 = 123;
String s4 = "789";
double d1 = 83.5;
char   c1 = 'A';
double d;

s = Integer.toString(n);                  // s = "123"
s = Double.toString(d1);                  // s = "83.5"
n = Integer.parseInt(s4);                 // n = 789
d = Double.valueOf(s4).doubleValue();     // d = 789.0
```

Graphics methods

The Graphics class is part of the java.awt package and includes the following methods. All of the following Graphics methods return void.

Selected Graphics methods

Method	Description
`drawPolygon(int[] xs, int[] ys, int n)`	Draws a closed polygon defined by arrays of x and y coordinates.
`drawLine(int x1, int y1, int x2, int y2)`	Draws a line between the points (x1, y1) and (x2, y2).
`drawOval(int x, int y, int w, int h)`	Draws the outline of an oval.
`drawRect(int x, int y, int w, int h)`	Draws the outline of the specified rectangle.
`drawString(String str, int x, int y)`	Draws the text given by the specified string, using the current font and colour.
`fillOval(int x, int y, int w, int h)`	Fills an oval bounded by the specified rectangle with the current colour.
`fillPolygon(int[] xs, int[] ys, int n)`	Fills a closed polygon defined by arrays of x and y coordinates.
`fillRect(int x, int y, int w, int h)`	Fills the specified rectangle.
`setColor(Color c)`	Sets the current colour.
`setFont(Font font)`	Sets the current font.

Math methods

The Math class contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions. Methods of the same name may be defined for different types (e.g. int, long, float, double).

Selected Math methods

Type	Method	Description
`int`	`abs(int a)`	Returns the absolute value of the argument.
`double`	`abs(double a)`	
`double`	`ceil(double a)`	Returns the smallest (closest to negative infinity) integer that is greater than or equal to the argument.
`double`	`floor(double a)`	Returns the largest (closest to positive infinity) integer that is less than or equal to the argument.
`int`	`max(int a, int b)`	Returns the greater of the two arguments.
`double`	`max(double a, double b)`	
`int`	`min(int a, int b)`	Returns the least of the two arguments.
`double`	`min(double a, double b)`	
`double`	`pow(double a, double b)`	Returns the value of the first argument raised to the power of the second argument.
`double`	`random()`	Generates a random number between 0.0 and 1.0
`long`	`round(double a)`	Returns the closest integer, with ties being rounded up.
`double`	`sqrt(double a)`	Returns the square root of the argument.

Examples

```
int a = (int)(Math.random()*4+2);   // a is set to one of 2, 3, 4, or 5
int b = Math.abs(-5);               // b = 5
int c = Math.abs(5);                // c = 5
double d = Math.pow(2,3);           // d = 8.0
double e = Math.max(-4,3);          // e = 3.0
double f = Math.floor(5.9);         // f = 5.0
double g = Math.ceil(5.9);          // g = 6.0
```

Event-driven programming

In *event-driven programming*, the flow of the program is determined by events such as user actions, including keyboard presses and mouse movements. Detecting an event and handling it is called *event handling*.

A *component* is a Graphical User Interface (GUI) object that enables the user to interact with the applet. There are many such components, including labels, text fields, and buttons. By default, components are placed in the applet window in the order they are added, from left to right, top to bottom.

Example – Button Count

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class ButtonCount extends Applet implements ActionListener
{
    Button enterButton;
    int count;

    public void init()
    {
        enterButton = new Button("Enter");
        add(enterButton);
        enterButton.addActionListener(this);
    }
    public void paint(Graphics g)
    {
        g.drawString("Number of button presses is "+ count, 20, 120);
    }

    public void actionPerformed(ActionEvent event)
    {
        count = count + 1;
        repaint();
    }
}
```

Methods

A *method* is a section of code that is referred to by name and can be called from elsewhere in a program. They enable code to be written once and used many times, including in other classes (e.g. Math.random(), g.drawString(), etc.).

Methods can take multiple arguments and optionally return a value.

How parameters are passed

A *parameter* is a variable in a method definition. An *argument* is the actual value of the variable that gets passed to a method. When variables are passed to a method, they are *passed by value* regardless of the original data type. This means that for each parameter, a copy is made. Consequently, whatever the method does to the argument, it has no effect on the original variable.

For example, in the following code, the method *change()* will return 11 but the original value of x = 10 is unchanged.

```
int x = 10;
int y = change(x);

public int change(int z){
    z++;
    return z;
}
```

Passing reference types as parameters

Passing a reference type (i.e. an array or other object) to a method is subtly, but significantly, different from passing a primitive data type. Although Java manipulates objects *by reference*, it still passes object references to methods *by value*, just like any other parameter. When an object is passed to a method, a copy of the *reference* is made, but not the object that the reference points to.

For example, in the following code, the *fillZero()* method will change the values in the array *table* to *0*.

```
int[] table = new int[8];
fillZero(table);

public void fillZero(int[] array) {
    for (int s=0; s<array.length; s++)
        array[s]=0;
}
```

Because strings are objects, when a string is passed to a method, the *reference* to the string is passed by value (a copy of the reference is passed). But because strings are immutable, changing the value of a string creates a *new string* that the *copy of the reference* now points to. In practice, we can think of strings as behaving like a primitive data type being passed by value.

Scope of variables

The *scope* of a variable is the part of the program where the variable is accessible.

- *Global variables* are declared just inside the class definition, but outside of any methods, and may be used within any method defined in the class.
- *Local variables* are declared within a method, constructor, or block (e.g. a loop), and can only be accessed within the part of the program where they have been declared. Method parameters can be considered local variables.

When a local variable has the same name as a global variable, the global variable is hidden within the scope of the local variable.

Objects and classes

Object-oriented programming is an approach to programming based on *objects* rather than functions.

- An *object* is a data structure that includes values (i.e. *variables*) and ways to manipulate data (i.e. *methods*).
- A *class* is a template for creating an object, including the initial values of any *variables* as well as *method* definitions.
- An object is an *instance* of a class.
- Any number of objects can be created (i.e. instantiated) from a class.

Important principles of object-oriented programming include encapsulation, inheritance, and polymorphism.

- *Encapsulation* – Encapsulation is the process of grouping related data and methods together in a class and makes *data hiding* possible. Variables (and methods) can be kept *private* with public getter and setter methods providing safe access to an object's data.
- *Inheritance* – A class can be derived from another class, inheriting its variables and methods. The class that inherits the properties of another is known as a *child class* (or subclass) and the class that is inherited from is known as a *parent class* (or superclass). For example, when a Java applet is created, it is a child of the parent Applet class.
- *Polymorphism* – Polymorphism is the ability of an object to take on many forms. Two examples of polymorphism in Java are *method overloading* (i.e. a class may have more than one method with the same name, as long as their argument lists are different) and *method overriding* (i.e. declaring a method in a child class that is also declared in a parent class).

Class definitions and constructors

A class has the following general structure:

```
<class modifiers> class <name of class>
{
    <variables declarations>
    <constructor declarations>
    <method declarations>
}
```

A *constructor* is a block of code similar to a method that is called when an object is *instantiated* (created).

- Constructors must have the same name as the class.
- Every class definition requires at least one constructor.
- Constructors never return a value (other than the object they have created), so they don't specify a return type (not even *void*).
- It is recommended to have a constructor that sets the instance variables to default values.
- Multiple constructor methods can be defined, but they still must all have the same name as the class and differ in the number and type of the parameters. This is known as *constructor overloading*.

Object creation

To use an object, you must first:

- *Declare* the object by stating its class and giving it a name. By convention, class names begin with a capital letter, and object names begin with lower case.
- *Instantiate* (and *initialise*) the object by calling one of the *constructors* for the object with the *new* operator.

For example, to create an Enter button:

```
Button myButton;
myButton = new Button("Enter");
```

Alternatively, an object can be declared and instantiated in one line:

```
Button myButton = new Button("Enter");
```

Example – Car class definition

```
public class Car
{
    private int maxSpeed;
    private String make;

    public Car(){                                       // constructor 1
        maxSpeed = 0;
        make = "";
    }

    public Car(int speed, String carMake){              // constructor 2
        maxSpeed = speed;
        make = carMake;
    }

    public void setMaxSpeed(int speed) {
        maxSpeed = speed;
    }

    public int getMaxSpeed(){
        return maxSpeed;
    }

    public void setMake(String carMake) {
        make = carMake;
    }

    public String getMake(){
        return make;
    }

} // end Car
```

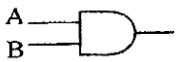
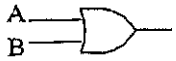
Example – Creation and use of Car objects

```
public class CarSample extends Applet
{
    Car    fredsCar, marysCar;

    public void init()
    {
        fredsCar = new Car();                           // using constructor 1
        fredsCar.setMaxSpeed(500);
        fredsCar.setMake("Honda");
        marysCar = new Car(600, "Porsche");             // using constructor 2
    }

    public void paint(Graphics g)
    {
        g.drawString("Fred's car speed " + fredsCar.getSpeed()+
                        " and make " + fredsCar.getMake(), 10, 20);
        g.drawString("Mary's car speed " + marysCar.getSpeed() +
                        " and make " + marysCar.getMake(), 10, 40);
    }
}
```

BOOLEAN LOGIC

Boolean logic (or Boolean algebra) is a branch of Mathematics where variables have two possible values, *true* or *false*, often denoted as *1* and *0* respectively.

Logic gates (Boolean operators)

Gate	Symbol	Logic circuit	Truth table	Description
AND	$\wedge$		A B $A \wedge B$ 0 0 0 0 1 0 1 0 0 1 1 1	output is true only if all the inputs are true
OR	$\vee$		A B $A \vee B$ 0 0 0 0 1 1 1 0 1 1 1 1	output is true only if one or more of the inputs are true
NOT	~		A ~A 0 1 1 0	output is true only if the input is false

Logic laws

Commutative laws

L1 $a \wedge b \equiv b \wedge a$

L2 $a \vee b \equiv b \vee a$

Associative laws

L4 $a \wedge (b \wedge c) \equiv (a \wedge b) \wedge c$

L5 $a \vee (b \vee c) \equiv (a \vee b) \vee c$

Law of negation

L6 $\sim \sim a \equiv a$

Distribution laws

L7 $a \vee (b \wedge c) \equiv (a \vee b) \wedge (a \vee c)$

L8 $a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$

De Morgan's laws

L10 $\sim(a \vee b) \equiv \sim a \wedge \sim b$

L11 $\sim(a \wedge b) \equiv \sim a \vee \sim b$

Idempotent laws

L12 $a \wedge a \equiv a$

L13 $a \vee a \equiv a$

Law of the excluded middle

L14 $a \vee \sim a \equiv T$

Law of contradiction

L15 $a \wedge \sim a \equiv F$

Laws of the constants

L21 $\sim T \equiv F$

L22 $\sim F \equiv T$

L23 $a \wedge T \equiv a$

L24 $a \wedge F \equiv F$

L25 $a \vee T \equiv T$

L26 $a \vee F \equiv a$

Complement rules of $\wedge$ and $\vee$

L27 $a \vee (b \wedge \sim a) \equiv a \vee b$

L28 $a \wedge (b \vee \sim a) \equiv a \wedge b$

Absorption laws

L33 $a \wedge (a \vee b) \equiv a$

L34 $a \vee (a \wedge b) \equiv a$

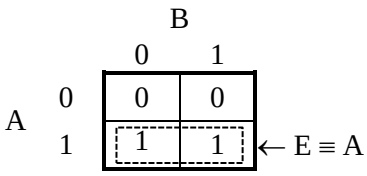
Karnaugh maps

A *Karnaugh map* is a visual method of simplifying a Boolean logic expression.

Example 1 – Two variables

Use a Karnaugh map to simplify the expression $E \equiv (A \wedge \sim B) \vee (A \wedge B)$

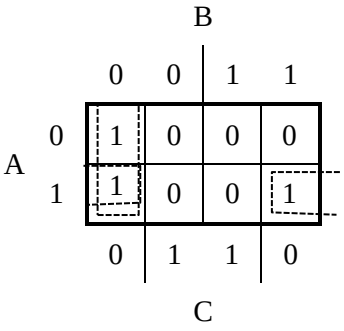
A	B	E
0	0	0
0	1	0
1	0	1
1	1	1



Example 2 – Three variables

This is a truth table for a logic device that has inputs A, B, and C, and a single output F. Use a Karnaugh map to determine a simple logic expression for F.

A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

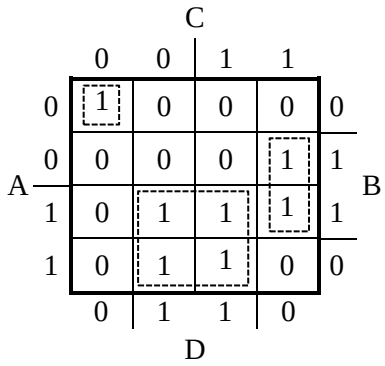


$F \equiv (\sim B \wedge \sim C) \vee (A \wedge \sim C)$

Example 3 – Four variables

This is a truth table for a logic device that has inputs A, B, and C, D, and a single output G. Use a Karnaugh map to determine a simple logic expression for G.

A	B	C	D	G
0	0	0	0	1
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1



$G \equiv (\sim A \wedge \sim B \wedge \sim C \wedge \sim D) \vee (A \wedge D) \vee (B \wedge C \wedge \sim D)$

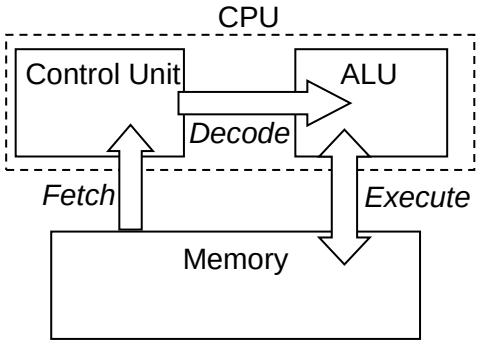
COMPUTER ARCHITECTURE

Computer architecture is a collection of rules and methods that describes how a set of hardware components and software technology interacts to form a computer system.

Structure of the Central Processing Unit (CPU)

Control unit	The control unit of the CPU contains circuitry that directs the entire computer system to carry out, or execute, stored program instructions. The control unit communicates with both the arithmetic/logic unit (ALU) and main memory.
Register	Registers are temporary storage areas for instructions or data. They are not a part of memory. They are special additional storage locations that offer the advantage of speed. Registers work under the direction of the control unit to accept, hold, and transfer instructions or data.
ALU	Stands for Arithmetic/Logic Unit. This is the part that executes the computer's commands including arithmetic operations and logical comparisons.
Memory	Memory is the part of the computer that holds data and instructions. Although closely associated with the central processing unit, memory is separate from it.
Bus	A bus is a collection of wires and connectors through which the data is transmitted. A bus is used to connect the components of the CPU and memory. The bus has two parts -- an address bus and a data bus. The data bus transfers data whereas the address bus transfers information about the data and where it should go.

THE MACHINE CYCLE	
FETCH	Get an instruction from memory and place in the instruction register within the control unit.
DECODE	Convert the instruction into computer commands that control the ALU and memory
EXECUTE	Process the commands using the ALU and memory



The TOY machine

TOY is an imaginary machine that models the structure of an early modern computer. The TOY machine has 16 *registers* (R[0] to R[F]), 256 words of *main memory* (00_{16} to FF_{16}), and 16 different instruction types (given by the *opcodes* 0 to F) and processes 16-bit words.

- All programs start at memory location 10.
- Memory address FF is connected to the input/output (e.g. keyboard/display).
- Register 0 (R[0]) is permanently set to value 0.

OPCODE	DESCRIPTION	FORMAT	PSEUDOCODE
0	halt	-	exit
1	add	1	R[d] ← R[s] + R[t]
2	subtract	1	R[d] ← R[s] - R[t]
3	and	1	R[d] ← R[s] & R[t]
4	xor	1	R[d] ← R[s] ^ R[t]
5	left shift	1	R[d] ← R[s] << R[t]
6	right shift	1	R[d] ← R[s] >> R[t]
7	load address	2	R[d] ← addr
8	load	2	R[d] ← mem[addr]
9	store	2	mem[addr] ← R[d]
A	load indirect	1	R[d] ← mem[R[t]]
B	store indirect	1	mem[R[t]] ← R[d]
C	branch zero	2	if (R[d] == 0) pc ← addr
D	branch positive	2	if (R[d] > 0) pc ← addr
E	jump register	-	pc ← R[d]
F	jump and link	2	R[d] ← pc; pc ← addr

Each TOY instruction consists of 4 hex digits (or 16 bits). The leading (left-most) hex digit encodes one of the 16 opcodes. The second hex digit refers to one of the 16 registers, which we call the destination register and denote by *d*.

The interpretation of the two rightmost hex digits depends on the opcode. With *Format 1* opcodes, the third and fourth hex digits are each interpreted as the index of a register. With *Format 2* opcodes, the third and fourth hex digits (the rightmost 8 bits) are interpreted as a memory address.

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>Format 1</td><td colspan="4">opcode</td><td colspan="4">destination (d)</td><td colspan="4">source (s)</td><td colspan="4">source (t)</td></tr>
<tr><td>Format 2</td><td colspan="4">opcode</td><td colspan="4">destination (d)</td><td colspan="8">address (addr)</td></tr>
</table>

Note: Instructions 0 and E have no format listed. Consider the following examples.
Instructions 0000, 0A14, 0BC7 would all halt execution. The last three hex digits are ignored.
Instructions E500, E514, E5C7 would all change the value of the program counter to the contents of register 5. The last two hex digits are ignored.

TOY programming and Java

Example – Addition

Write TOY code equivalent to the following Java code.

```
int x = 6 + 8;
```

To perform addition, we load the values to be added into registers and use opcode 1.

Memory Address	Contents	Pseudocode	Explanation
00	0006	data	($0000\ 0000\ 0000\ 0110_2$, 6_{10})
01	0008	data	($0000\ 0000\ 0000\ 1000_2$, 8_{10})
05	0000	data	location chosen for variable x
10	8A00	R[A] ← mem[00]	load 6 into register A
11	8B01	R[B] ← mem[01]	load 8 into register B
12	1CAB	R[C] ← R[A] + R[B]	add 6 and 8 and store result (14) in register C
13	9C05	mem[05] ← R[C]	copy result (14) to variable x
14	0000	halt	stop execution

Example – Multiplication

Write TOY code equivalent to the following Java code.

```
int x = 3 * 8;
```

Multiplication can be performed by either:

- left shifts (each shift multiplies by 2), e.g. $x = x \times 10$ could be implemented as $x = x \times 8 + x \times 2$
- repeated addition, e.g. $x = x \times 10$ could be implemented by adding x to itself 10 times

Similarly, division can be performed using right shifts or repeated subtraction.

Memory Address	Contents	Pseudocode	Explanation	
01	0003	data	($0000\ 0000\ 0000\ 0011_2$, 3_{10})	
02	0008	data	($0000\ 0000\ 0000\ 1000_2$, 8_{10})	
03	0000	data	($0000\ 0000\ 0000\ 0000_2$, 0_{10})	
04	0001	data	($0000\ 0000\ 0000\ 0001_2$, 1_{10})	
05	0000	data	location chosen for variable x	
10	8A01	R[A] ← mem[01]	load 3 into register A	a = 3;
11	8B02	R[B] ← mem[02]	load 8 into register B	b = 8;
12	8C03	R[C] ← mem[03]	load 0 into register C	c = 0;
13	8104	R[1] ← mem[04]	load 1 into register 1	// always 1
14	CA18	if (R[A]==0) pc ← 18	if R[A] == 0 exit the loop	while (a!=0) {
15	1CCB	R[C] ← R[C] + R[B]	add R[C] to R[B]	c = c + b
16	2AA1	R[A] ← R[A] – R[1]	decrement R[A]	a = a - 1
17	C014	pc ← 14	go to 14	}
18	9C05	mem[05] ← R[C]	copy result (24) to variable x	
19	0000	halt	stop execution	

Example – If-else

To implement an *if-else* statement in TOY, use opcode C (branch zero) or opcode D (branch positive) and jump over the blocks of code that don't apply.

Java example	Pseudocode (Java version)	Pseudocode (TOY version)

`if (x==3)`
`  x=x+3;`
`else`
`  x=x-1;`
	`if (a==b)`
	`  <code 1>`
	`else`
	`  <code 2>`
		`if (a-b=0) branch to code1`
		`  <code 2>`
		`branch to next`
		`  <code 1>`
		`next`

Memory Address	Contents	Pseudocode	Explanation
00	0000	data	variable x
01	0003	data	constant 3
02	0001	data	constant 1
10	8A00	R[A] ← mem[00]	load x into register A
11	8B01	R[B] ← mem[01]	load constant 3 into register B
12	2CAB	R[C] ← R[A] - R[B]	form test for x-3==0
13	CC18	if (R[C]==0) pc ← 18	if x==3 go to 18 (code 1)
14	8B02	R[B] ← mem[02]	code 2: load constant 1 into register B
15	2CAB	R[C] ← R[A] – R[B]	take 1 off x and store result in register C
16	9C00	mem[00] ← R[C]	store register C in x (i.e. x=x-1)
17	C01B	if (R[0]==0) pc ← 1B	go to 1B (cont)
18	8B01	R[B] ← mem[01]	code 1: load constant 3 into register B
19	1CAB	R[C] ← R[A] + R[B]	add 3 to x and store result in register C
1A	9C00	mem[00] ← R[C]	store register C in x (i.e. x=x+3)
1B	0000	halt	stop execution (or continue program from here)

Example – While / for loop

Remember that *for* loops can be implemented as *while* loops

Java while loop	Java for loop	Pseudocode (TOY version)

`x = 0;`
`while(x<5)`
`{`
`  // code`
`  x++;`
`}`
	`for (int x=0; x<5; x++)`
	`{`
	`  // code`
	`}`
		`x = 0`
		`// start of loop`
		`if (x-4 > 0) goto next // since x<5`
		`  // code`
		`  x = x+1`
		`  // goto start of loop`
		`next`

Memory Address	Contents	Pseudocode	Explanation
00	0000	data	constant 0 (loop first value)
01	0004	data	constant 4 (loop last value) (since condition is <5)
02	0001	data	constant 1 (loop increment)
03	0000	data	variable
10	8A00	R[A] ← mem[00]	load initial value for x into register A
11	8B01	R[B] ← mem[01]	load constant 4 into register B
12	8D02	R[D] ← mem[02]	load constant 1 into register D
13	2CAB	R[C] ← R[A] – R[B]	load x-4 into register C
14	DC18	if (R[C]>0) pc ← 18	go to 18 (next) if loop finished
15	1AAD	R[A] ← R[A] + R[D]	store x+1 in register A
16	9A03	mem[03] ← R[A]	store value of x in memory
17	C013	if (R[0]==0) pc ← 13	go to 13
18	0000	halt	stop execution (or continue program from here)

DATA REPRESENTATION

Data representation deals with how data is represented and structured symbolically for storage and communication in a computer system.

Representation of boolean

A boolean variable has the two possible values, *true* and *false*. This means a boolean value can be represented by a single *bit* (binary digit), which has two possible values, *0* (false) and *1* (true).

Representation of numbers

Computers store numbers (and indeed all data) using a binary representation. Whereas the decimal (base 10) number system uses 10 digits (0,1,2,3,4,5,6,7,8,9), binary (base 2) is a system for representing numbers using two digits (0, 1). Any number can be represented as a sequence of bits.

Unsigned binary

In unsigned binary, each bit represents an increasing power of 2 as shown in this table:

2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
256	128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625	0.03125

- A sequence of bits is referred to as a *word*.
- A *byte* is an 8-bit word and a *nibble* is a 4-bit word.
- The left-most bit of a word is known as the *most significant bit* and the right-most bit is the *least significant bit*.
- One *byte* (ie. an 8-bit word) can represent numbers ranging from $0000\ 0000_2 = 0_{10}$ to $1111\ 1111_2 = 255_{10}$
- The largest integer in a *n*-bit unsigned representation is $2^n - 1$ (e.g. the largest number in a 16-bit unsigned representation is $2^{16} - 1 = 65535$).

Example – Binary to decimal conversion

$101_2 \Rightarrow 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 4 + 0 + 1 = 5_{10}$

$1100_2 \Rightarrow 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 8 + 4 = 12_{10}$

$0.101_2 \Rightarrow 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} = 0.5 + 0.125 = 0.625_{10}$

Example – Decimal to binary conversion

To convert a decimal integer to binary, we can repeatedly divide by 2, recording the remainders as shown:

e.g. convert 211_{10} to binary.

Dividends	Remainders	Notes
2 |211		
2 |105	1	Successive division continues until the quotient becomes zero.
2 |52	1	
2 |26	0	
2 |13	0	The binary number is determined by reading the remainders from last to first.
2 |6	1	
2 |3	0	
2 |1	1	Hence $211_{10} = 1101\ 0011_2$
|0	1	

Example – Decimal fraction to binary conversion

To convert a decimal fraction to binary, we can repeatedly multiply by 2, recording the whole number (i.e. 0 or 1) and fraction parts separately as shown:

Convert 5.2_{10} to binary.

$5_{10} = 101_2$

$0.2 \times 2 = 0 + 0.4$	*first digit = 0*
$0.4 \times 2 = 0 + 0.8$	*second digit = 0*
$0.8 \times 2 = 1 + 0.6$	*third digit = 1*
$0.6 \times 2 = 1 + 0.2$	*fourth digit = 1*
$0.2 \times 2 = 0 + 0.4$	*fifth digit = 0*
$0.4 \times 2 = 0 + 0.8$	*sixth digit = 0*
etc.	

Writing as a recurring binary number:

$0.2_{10} = 0.001100110011..._2$

Rounding off to, say, 5 binary places:

$0.2_{10} = 0.00110_2$

Therefore, $5.2_{10} = 101.00110_2$ (to 5 binary places)

Notes

Convert the whole number part separately.

Starting with the fraction part of the original number, repeatedly multiply by 2.

The process continues until enough digits have been obtained or the fractional part becomes zero.

An exact decimal fraction may not have an exact binary equivalent. In that case, go to one more place than asked for, then round.

For example, the decimal number 0.2 cannot be represented exactly in binary. It must be rounded off to a finite number of binary places, giving rise to round-off error.

Two's complement

Two's complement is a common method of representing signed numbers. The two's complement of a binary number is its complement with respect to 2^n where *n* is the number of bits in the word. In practice, the *two's complement* of a number is its inverse.

Example – Taking the two's complement (Method 1)

1. Take the *one's complement* by inverting all the bits.
2. Add *1*.

```
        0010 1010    42
  →     1101 0101
      + 0000 0001
  →     1101 0110   -42
```

Example – Taking the two's complement (Method 2)

1. Work from right to left, starting with the least significant bit.
2. Copy any *0* bits until the first *1* bit and copy the first *1* bit.
3. Invert all subsequent bits.

```
        0010 1010    42
  →     1101 0110   -42
```

Example – Converting a two's complement binary number to decimal

To interpret a number in two's complement representation:

1. If the most significant bit is *0*, the number is positive (or zero).
   → Convert the number to decimal normally.
2. If the most significant bit is *1*, the number is negative.
   → Take the two's complement to find the number's inverse.
   (Note: Because the original number was negative, the inverse must be a positive number.)
   → Convert the (positive) binary number to decimal normally.
   → Invert the decimal number to determine the value of the original (negative) binary number.

Convert the following 8-bit two's complement numbers to decimal:

a) 0000 0011

The most significant bit is 0, so it is a positive number.

Therefore, $0000\ 0011_2 = 3_{10}$

b) 1111 0001

The most significant bit is 1, so it is a negative number. Take the two's complement before converting to decimal.

```
   1111 0001
→ 0000 1111  → 15
```

Therefore, $1111\ 0001_2 = -15_{10}$

Binary arithmetic

Arithmetic in binary is very similar to decimal arithmetic.

```
  1101      13
+ 1110    + 14
 11011      27
```

```
1 + 0 → 1
0 + 1 → 1
1 + 1 → 0, carry 1
1 + 1 + 1 → 1, carry 1
```

An *overflow* occurs when there aren't enough bits to represents the result of an operation.

An advantage of two's complement representation is that it eliminates the need for a separate subtraction operation. This is because subtracting a number is equivalent to adding its opposite (which is found by *taking the two's complement).*

Example – Subtraction using two's complement representation

To carry out the operation *7 – 5* using an 8-bit two's complement representation, we first take the two's complement of 5 to find -5, and then calculate *7 + (-5)*.

```
 7        0000 0111   becomes…     0000 0111
-5      - 0000 0101                + 1111 1011
 2                                   0000 0010  = 2
```

Hexadecimal

Hexadecimal (hex, base 16) provides a compact way of representing binary numbers.

Because a 4-bit word can represent 16 possible values ($2^4 = 16$), it can be also represented by a single hex digit.

Example – Binary to hexadecimal conversion

A simple way to convert a binary number to hexadecimal is to split the binary number into 4-bit words, then convert each word to its hex equivalent.

Convert 01001111010101_2 into hexadecimal.

01001111010101_2
=> $01\ 0011\ 1101\ 0101_2$
=> $1\ 3\ D\ 5_{16}$

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Floating-point representation

Floating-point representation is a method of storing numbers in which a computer word is divided three parts that represent the sign, the exponent, and the mantissa.

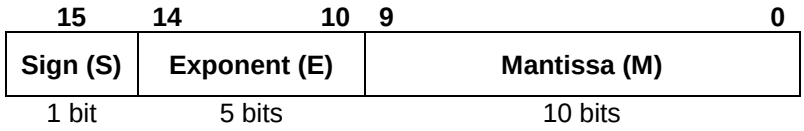
$\pm M \times 2^E$ where M is the mantissa
E is the exponent in two's complement

Floating-point representations support a trade-off between *range* (determined by the number of bits used to represent the exponent) and *precision* (determined by the number of bits used to represent the mantissa).

Example – A 16-bit floating-point representation

Consider a 16-bit floating-point representation in which:

- The most significant bit is the sign bit, with 0 for positive numbers and 1 for negative numbers.
- The following 5 bits represent the exponent in 2s complement.
- The remaining 10 bits represent the mantissa, where 0 ≤ mantissa < 1



- The largest possible number using this format is: 0 01111 1111111111 → $0.1111111111 \times 2^{1111\ \{15\}}$
- The smallest positive number is: 0 10000 1000000000 → $0.1 \times 2^{-10000\ \{-16\}}$

Using this representation, interpret the following 16-bit words as decimal numbers.

a) 0 11100 1100000000

sign = 0 → positive (+)
exponent = 11100 → - 4
mantissa = 0.11 → 0.5 + 0.25 = 0.75

decimal value $= + 0.75 \times 2^{-4}$
$= + 0.046875$

b) 1 00111 1010000000

sign = 1 → negative (-)
exponent = 00111 → 7
mantissa = 0.101 → 0.5 + 0.125 = 0.625

decimal value $= - 0.625 \times 2^{7}$
$= - 80$

Representation of characters

Characters are represented using an encoding system in which each character is represented by a number, stored as a binary word. Two of the most common character encoding systems are ASCII and Unicode.

ASCII

American Standard Code for Information Interchange (ASCII) is the system that most modern character encoding systems are based on. It was originally based on English and encodes 128 characters into 7-bit integers, but numerous 8-bit variations of ASCII have been developed. These typically preserve the original ASCII character mapping and add extra character definitions after the first 128 characters.

Unicode

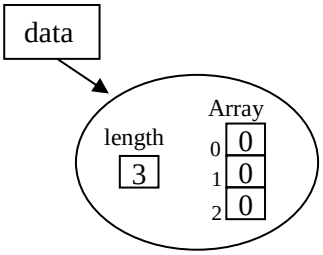
Unicode is the character encoding system used by Java. It includes the standard ASCII character set as its first 256 characters (with the high byte set to 0), but also includes several thousand other characters representing most international alphabets. The Unicode standard was initially designed using 16 bits to encode characters. The numerical values are unsigned 16-bit values between 0 and 65535 (65536 characters in total).

Representation of arrays

Consider the following definition for an array:

```
int[] data = new int [3];
data[0] = 0;
data[1] = 0;
data[2] = 0;
```

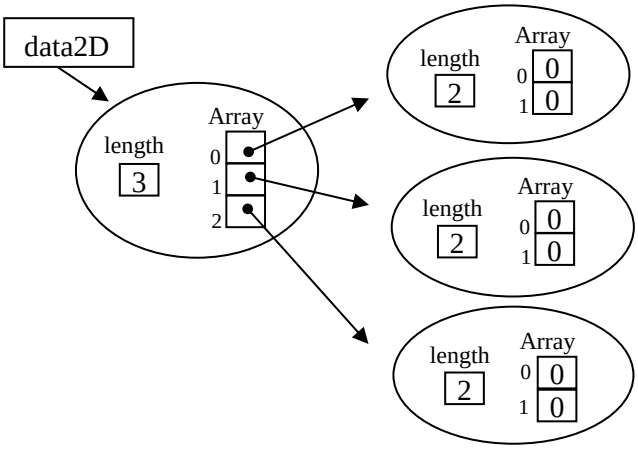
This can be represented diagrammatically as follows:



Consider the following definition for a two-dimensional array:

```
int[][] data2D = new int [3][2];
```

This can be represented diagrammatically as follows (assuming array has been initialised to all zeroes):



Representation of images

One way to represent an image is to consider it as a collection of *pixels* (picture elements). An image that is 400 pixels wide and 300 pixels high has a total of 400 × 300 = 120,000 pixels.

Black and white images only need one bit per pixel. For example, an uncompressed black and white 400 x 300 image requires 15,000 bytes (120,000 bits/8).

Colour pictures need additional bits to represent the range of possible colours. *Colour depth* is the number of bits used to represent the colour of a single pixel. For example, a 16-bit colour system supports up to 2^{16} = 65,536 colours.

An uncompressed 16-bit 400 × 300 image requires 240,000 bytes (400 × 300 × 16 = 1,920,000 bits/8).

Representation of sounds

Sampling is the process of turning analogue sound waves into digital (binary) signals. The system samples the sound by taking snapshots of its amplitude at regular intervals. In general, sampling should happen at twice the highest frequency to be recorded. Sample depth is the number of bits used to encode each sample. In general, the higher the sample rate and sample depth, the more accurately the digital sound reflects its real-life source and the larger the disk space required to store it.

For example, a 2-minute audio file sampled at a rate of 44.1KHZ and a depth of 16 bits requires 10,584,000 bytes (120 × 44100 × 16 = 84,672,000bits/8).

Data compression

Data compression is the process of encoding information using fewer bits than an unencoded representation would use.

Lossless compression schemes are reversible, which means that the original data can be reconstructed. Lossy schemes accept some loss of data in order to achieve higher compression.

ASCII table

Dec	Char	Binary	Dec	Char	Binary	Dec	Char	Binary
0	NUL	0000 0000	48	0	0011 0000	96	`	0110 0000
1	SOH	0000 0001	49	1	0011 0001	97	a	0110 0001
2	STX	0000 0010	50	2	0011 0010	98	b	0110 0010
3	ETX	0000 0011	51	3	0011 0011	99	c	0110 0011
4	EOT	0000 0100	52	4	0011 0100	100	d	0110 0100
5	ENQ	0000 0101	53	5	0011 0101	101	e	0110 0101
6	ACK	0000 0110	54	6	0011 0110	102	f	0110 0110
7	BEL	0000 0111	55	7	0011 0111	103	g	0110 0111
8	BS	0000 1000	56	8	0011 1000	104	h	0110 1000
9	HT	0000 1001	57	9	0011 1001	105	i	0110 1001
10	LF	0000 1010	58	:	0011 1010	106	j	0110 1010
11	VT	0000 1011	59	;	0011 1011	107	k	0110 1011
12	FF	0000 1100	60	<	0011 1100	108	l	0110 1100
13	CR	0000 1101	61	=	0011 1101	109	m	0110 1101
14	SO	0000 1110	62	>	0011 1110	110	n	0110 1110
15	SI	0000 1111	63	?	0011 1111	111	o	0110 1111
16	DLE	0001 0000	64	@	0100 0000	112	p	0111 0000
17	DC1	0001 0001	65	A	0100 0001	113	q	0111 0001
18	DC2	0001 0010	66	B	0100 0010	114	r	0111 0010
19	DC3	0001 0011	67	C	0100 0011	115	s	0111 0011
20	DC4	0001 0100	68	D	0100 0100	116	t	0111 0100
21	NAK	0001 0101	69	E	0100 0101	117	u	0111 0101
22	SYN	0001 0110	70	F	0100 0110	118	v	0111 0110
23	ETB	0001 0111	71	G	0100 0111	119	w	0111 0111
24	CAN	0001 1000	72	H	0100 1000	120	x	0111 1000
25	EM	0001 1001	73	I	0100 1001	121	y	0111 1001
26	SUB	0001 1010	74	J	0100 1010	122	z	0111 1010
27	ESC	0001 1011	75	K	0100 1011	123	{	0111 1011
28	FS	0001 1100	76	L	0100 1100	124	|	0111 1100
29	GS	0001 1101	77	M	0100 1101	125	}	0111 1101
30	RS	0001 1110	78	N	0100 1110	126	~	0111 1110
31	US	0001 1111	79	O	0100 1111	127	DEL	0111 1111
32	Space	0010 0000	80	P	0101 0000			
33	!	0010 0001	81	Q	0101 0001			
34	"	0010 0010	82	R	0101 0010			
35	#	0010 0011	83	S	0101 0011			
36	$	0010 0100	84	T	0101 0100			
37	%	0010 0101	85	U	0101 0101			
38	&	0010 0110	86	V	0101 0110			
39	'	0010 0111	87	W	0101 0111			
40	(	0010 1000	88	X	0101 1000			
41	)	0010 1001	89	Y	0101 1001			
42	*	0010 1010	90	Z	0101 1010			
43	+	0010 1011	91	[	0101 1011			
44	,	0010 1100	92	\	0101 1100			
45	-	0010 1101	93	]	0101 1101			
46	.	0010 1110	94	^	0101 1110			
47	/	0010 1111	95	_	0101 1111			