Computer Programming

CS 1 Introduction to Computers and Computer Technology
Rick Graziani

Fall 2017
Algorithm

- An algorithm is an ordered set of unambiguous, executable steps that defines a terminating process.
- Algorithms are part of many activities, even mundane ones.
- Note: Researchers believe that the human mind including imagination, creativity, and decision making, is actually the result of algorithm execution.
  - This is used in artificial intelligence
Example

- Obtain a basket of unshelled peas and an empty bowl.
- As long as there are unshelled peas in the basket continue to execute the following steps:
  a. Take a pea from the basket
  b. Break open the pea pod
  c. Dump the peas from the pod into the bowl
  d. Discard the pod
Defining the Algorithm

Non-terminating sequence

1 2 3 4 5 6 7 8 9 10 11 … (this could go on for ever!)

Ambiguous

Organize the CDs  (By title? By artist? By genre?)

• An **algorithm** is an ordered set of unambiguous, executable steps that defines a terminating process.
  – Steps do not have to be executed in sequence.

• **Non-Terminating Sequence:**
  – Make a list of positive integers.
  – The above requirement could not be performed in an algorithm, because it does not terminate (it is infinite).

• **Unambiguous**
  – The instructions must be clear, specific and direct
  – No room for creativity or interpretation
Abstract Nature of Algorithms

• An algorithm can represented in several ways.
• Example: Algorithm to convert temperatures from Celsius to Fahrenheit:
  – As an algebraic formula:
    • $F = \frac{9}{5}C + 32$
  – As a written instruction:
    • Multiply the temperature reading in Celsius by 9/5 and then add 32
Algorithm Representation

• Algorithm requires some form of a language.

• **Algorithm is a form of communication**
  – Don’t want misunderstandings
  – Proper level of detail
  – Proper level of difficulty

• Problems arise when:
  – Steps not precisely defined
  – Not enough detail
Algorithm Representation

<table>
<thead>
<tr>
<th>Op-code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOAD reg. R from cell XY.</td>
</tr>
<tr>
<td>2</td>
<td>LOAD reg. R with XY.</td>
</tr>
<tr>
<td>3</td>
<td>STORE reg. R at XY.</td>
</tr>
<tr>
<td>4</td>
<td>MOVE R to S.</td>
</tr>
<tr>
<td>5</td>
<td>ADD S and T into R. (2’s comp.)</td>
</tr>
<tr>
<td>6</td>
<td>ADD S and T into R. (floating pt.)</td>
</tr>
<tr>
<td>7</td>
<td>OR S and T into R.</td>
</tr>
<tr>
<td>8</td>
<td>AND S and T into R.</td>
</tr>
<tr>
<td>9</td>
<td>XOR S and T into R.</td>
</tr>
<tr>
<td>A</td>
<td>ROTATE reg. R X times.</td>
</tr>
<tr>
<td>B</td>
<td>JUMP to XY if R = reg. 0.</td>
</tr>
<tr>
<td>C</td>
<td>HALT.</td>
</tr>
</tbody>
</table>

- **Primitive** – A well defined set of building blocks (terms) used in computer science.
  - Arithmetic and logic operations built into the language
  - Removes any ambiguity
  - Includes its own syntax
- **Programming language** – A collection of primitives (terms) and the rules that state how the primitives can be combined to represent more complex ideas.
Machine language uses primitives

High-level programming languages (C++, Java) use higher-level primitives, constructed from the lower-level machine language primitives.

This results in an easier set of instructions to write.

More later.
Pseudocode

1. Enter two numbers.
2. Add the numbers together.
3. Display the result.

- **Pseudocode** – A notational system in which ideas can be expressed informally during the algorithm development process. (written sentence)
- **Used independently** of the programming language.
  - Each programming language has its own primitives and rules.
Pseudocode

Conditional selection

• The selection of one of two possible activities depending upon the truth or falseness of some condition

  \[
  \text{if condition then action}
  \]

  or

  \[
  \text{if condition then (activity)}
  \]

  \[
  \text{else (activity)}
  \]

• If this condition is true, perform this activity.

  \[
  \text{If (sunny)}
  \]

  \[
  \text{then (put on sunscreen)}
  \]

• If this condition is true, perform this activity, otherwise perform a different activity.

  \[
  \text{If (sunny)}
  \]

  \[
  \text{then (go swimming)}
  \]

  \[
  \text{else (go bowling)}
  \]
Another common semantic structure is the repeated execution of a statement or sequence of statements as long as some condition remains true.

while condition do activity

Also known as a while loop

Examples:

while (tickets remain to be sold) do (sell a ticket)
Repeating structure

Counter

Hello
Hello
Hello
Hello
Hello

<End of loop>

Task: Write Hello 500 times.

Pseudocode
Counter = 1
While counter is less than or equal to 500, write the word “Hello” and add 1 to Counter.

Programming Code
Counter = 1
While (counter <= 500)
do
(print “Hello”;
Counter ← Counter + 1)
For loop

Counter  Hello
         Hello
         Hello
         Hello
         …
         Hello

- A for loop can be used to accomplish the same thing as a while loop.
- Note: There are some differences between while and for loops.

Counter = 1
For (Counter <= 500)
  do
    (print the message “Hello”;
     Counter ← Counter + 1)
For loop

- A for loop can be used to accomplish the same thing as a while loop.
- Note: There are some differences between while and for loops.

For (Counter = 1; Counter <= 500; Counter ← Counter + 1)
do
(print the message “Hello”)

Counter  Hello
501  Hello
Hello
Hello
Hello
Hello
...
Programming Concepts

- Program consists of:
  - **Declarative statements**
    - Variables and data types
  - **Imperative statements**
    - Procedures, instructions, and algorithms
  - **Comments**
    - Enhance readability
    - Used throughout the program
Variables and Data Types

- **Variable** – A location in RAM (main memory), given a descriptive name, which stores information.
- **Data type** – Variables are a type of day which can be:
  - **Number**
    - Integer: 0, 1, 2, 3, 4, 5, 6, etc.
    - Real (floating point): 9.75, 300.5412
  - **Character**: a, b, c, A, B, C, etc.
  - **String** of text: “123 Main Street”
- **Working area or scratch pad for the program.**
Variables are often associated with a data structure.

**Data structure** – A conceptual shape of arrangement of data

**Homogeneous Array**
- Block of values of the same type
- Such as a one dimensional list or a two dimensional array (row, column)

**Example: String or array of characters**
- `char Last_Name[25]`
Data structure

### Frame

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dan</strong></td>
<td>3</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Sue</strong></td>
<td>12</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Gary</strong></td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mary</strong></td>
<td>7</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Integer pins [bowler, frame] pins [Mary, 2] = 7

- Two dimensional array
  - Row and column
  - Integer pins [bowler, frame]
Assigning Value

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>125</td>
</tr>
<tr>
<td>Float</td>
<td>9.75</td>
</tr>
<tr>
<td>Character</td>
<td>d</td>
</tr>
</tbody>
</table>

**Assignment statement** – Statement which assigns a value to a variable.
- Value assigned from **user input**
- Value assigned from **another variable**
- Value assigned from a **specific value**
- Value assigned from a **calculation** that can include both variables and assigned values.

Fahrenheit = \((9/5)\text{Celcius} + 32\)
#include <stdio.h>

This is a preprocessor command that includes standard input output header file (stdio.h) from the C library before compiling a C program.

```c
int main()
```

This is the main function from where execution of any C program begins.

```c
{
This indicates the beginning of the main function.

```c
/*_some_comments_*/
```

whatever is given inside the command “/*   */” in any C program, won’t be considered for compilation and execution.

```c
printf("_____________");
```

printf command prints the output onto the screen.

```c
getch();
```

This command waits for any character input from keyboard.

```c
return 0;
```

This command terminates C program (main function) and returns 0.

```c
}
```

This indicates the end of the main function.
# This is a comment in the program.
# This is used for documentation

print("This will print whatever I write here.")
This will print whatever I write here.

# Display arithmetic calculations
print(2 + 3)
5

value = 2 + 3
print(value)
print(value)
5
Print("The sum of 2 + 3 is", value)
The sum of 2 + 3 is 5
# This ask the user to enter a value
value1 = input("What is your first number: ")
What is your first number: 9

# This ask the user to enter a second value
value2 = input("What is your second number: ")
What is your second number: 6

total = int(value1) + int(value)

print("The total is ", total)
The is 15
print ("Hello world!")

apples-mbp-9:Python rigrazia$ python3 1-hello.py
Hello world!
apples-mbp-9:Python rigrazia$
value1 = 10
value2 = 5

print(value1 + value2)

print("\nBy storing the sum in a variable I can refer to the sum any time I want."")
print("\nI don't have to do the calculation every time.")

total = int(value1) + int(value2)
print("\nThe sum of value1 + value2 is ", total)
print("Here is the total again", total)
print(total,"is the total!")
print("Welcome to my adder program!")
print("I will ask you for two integers and give you the sum")

value1 = input("What is your first number: ")
value2 = input("What is your second number: 

total = int(value1) + int(value2)
print("Your total is", total)

apples-mbp-9:Python rigrazia$ python3 3-enter-values.py
Welcome to my adder program!

I will ask you for two integers and give you the sum
What is your first number: 10
What is your second number: 3

Your total is 13
apples-mbp-9:Python rigrazia$
print("Welcome to my adder program for real numbers!")
print("I will ask you for two real numbers and give you the sum")

value1 = input("What is your first number: ")
value2 = input("What is your second number: ")

total = float(value1) + float(value2)
print("Your total is", total)
print("Are you older than Rick?")
print("\nI will ask you your age and tell you if you're older than Rick")

YourAge = input("\nWhat is your age: ")

if int(YourAge) < 60:
    print("You are younger than Rick")
else:
    print("You are older or the same age as Rick")

apples-mbp-9:Python rigrazia$ python3 5-Age-1.py
Are you older than Rick?

I will ask you your age and tell you if you're older than Rick

What is your age: 21
You are younger than Rick
apples-mbp-9:Python rigrazia$
print("Are you older than Rick?")
print("I will ask you your age and tell you if you're older than Rick")

YourAge = input("What is your age: ")

if int(YourAge) < 60:
    print("You are younger than Rick")
elif int(YourAge) > 60:
    print("You are older than Rick")
else:
    print("You are the same age as Rick")

apples-mbp-9:Python rigrazia$ python3 6-Age-2.py
Are you older than Rick?

I will ask you your age and tell you if you're older than Rick

What is your age: 60
You are the same age as Rick
value1 = 10
value2 = 5

print(value1 + value2)

print("By storing the sum in a variable I can refer to the sum any time I want."")
print("I don't have to do the calculation every time."")

total = int(value1) + int(value2)
print("The sum of value1 + value2 is ", total)
print("Here is the total again", total)
print(total,"is the total!")

apples-mbp-9:Python rigrazia$ python3 2-add.py
15

By storing the sum in a variable I can refer to the sum any time I want.

I don't have to do the calculation every time.

The sum of value1 + value2 is  15
Here is the total again 15
15 is the total!
apples-mbp-9:Python rigrazia$
print("\nRick wants me to write on the whiteboard.... 'I will not throw paper airplanes in class 10 times"")

for NumberTimes in range(0, 10):
    print("I will not throw paper airplanes in class")

Rick wants me to write on the whiteboard.... 'I will not throw paper airplanes in class 10 times
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
apples-mbp-9:Python rigrazia$
Rick wants me to write on the whiteboard.... 'I will not throw paper airplanes in class"

maximum = input("How many times do you need to write this? ")

for NumberTimes in range(0, int(maximum)):
    print("I will not throw paper airplanes in class")

Rick wants me to write on the whiteboard.... 'I will not throw paper airplanes in class

How many times do you need to write this? 5
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
I will not throw paper airplanes in class
apples-mbp-9:Python rigrazia$
• “I want you to write on the chalk board, ‘I will not throw paper airplanes in class’, 500 times.”

```c
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```
#include <stdio.h>

int main(void)
{
    int count;

    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.\n");

    return 0;
}

Variable Count which is type integer

I will not throw paper airplanes in class.
I will not throw paper airplanes in class.
I will not throw paper airplanes in class.

...
Variables

- Variables are declared, defined in the declaration section of the program.
  - Usually at the beginning of the program
  - Examples:
    - `int height_in_inches`
    - `char first_initial`
    - `float price`
Variables

- Can be given an initial value within the program
- Value may change from:
  - Program instructions
  - User input

```c
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
    {
        printf("I will not throw paper airplanes in class.\n");
    }
    return 0;
}
```

I will not throw paper airplanes in class.
I will not throw paper airplanes in class.
I will not throw paper airplanes in class.

...  
I will not throw paper airplanes in class.
I will not throw paper airplanes in class.
Variable Count which is type integer

```
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

- Count = 1
- For Loop – As long as Count is less than or equal to 500
  - Add 1 to Count
  - Print “I will not throw paper airplanes in class”
- Exit program
Software Development
Definitions

Software or Program
Instructions that tell the computer what to do

Programmer
Someone who writes computer programs
# CPU Instruction Set

A vocabulary (list) of instructions which can be executed by the CPU

- The **only** instructions the CPU can run or execute
- Example of a CPU’s Instruction Set

### Instruction Set

<table>
<thead>
<tr>
<th>Instruction Set</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>Move</td>
</tr>
<tr>
<td>0010</td>
<td>Compare</td>
</tr>
<tr>
<td>0011</td>
<td>Bit test</td>
</tr>
<tr>
<td>0100</td>
<td>Bit clear</td>
</tr>
<tr>
<td>0101</td>
<td>Bit set</td>
</tr>
<tr>
<td>0110</td>
<td>Add</td>
</tr>
<tr>
<td>0111</td>
<td>See group 10</td>
</tr>
<tr>
<td>1000</td>
<td>See groups 11, 13, 14</td>
</tr>
<tr>
<td>1001</td>
<td>Move byte</td>
</tr>
</tbody>
</table>
First Generation Languages
(Machine Language)

- Programming computers using the CPU’s instruction set
- Also known as **Machine Language**
- **(timeline)**

**Machine Code File**
A software file which contains the instructions from the CPU’s instruction set.
First Generation Languages
(Machine Language)

Advantages of First Gen.
• Software programs execute (run) relatively very quickly
• Software programs are relatively small in size
• (Insignificant advantages today)

Disadvantages of First Gen.
• Difficult to write, very detailed and takes a long time
• Difficult to read
• Difficult to debug
debug = the process to find mistakes in a software program
Second Generation Languages (Assembly Language)

Assembly Language = The English-like instructions which are equivalent to the CPU’s instruction set

Source Code= The actual instructions written by a programmer

Compiler = Software which translates source code instructions of a particular language into machine code
Assembly languages have direct, one-to-one mapping to machine instructions. But, a single line of a high-level programming language might result in dozens of instructions being executed by the CPU.
Second Generation Languages (Assembly Language)

**Question:** Which of these two files (source code file or machine code file) will the user need to run this software program?

**Advantages** of Second Gen.
- Easier to read than first gen.
- Easier to write than first gen.
- Easier to debug than first gen.

**Disadvantages** of Second Gen.
- Still very difficult to write programs
Using a compiler
Grace Hopper wrote the first compiler program
Using an Interpreter

Write a Program → Compiling → Running

Java Files → Compiler → Interpreter

Public Class
System

class File
101010101
010101010

C:\Java\
## Interpreter versus Compiler

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translates program one statement at a time.</td>
<td>Scans the entire program and translates it as a whole into machine code.</td>
</tr>
<tr>
<td>It takes less amount of time to analyze the source code but the overall execution time is slower.</td>
<td>It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.</td>
</tr>
<tr>
<td>No intermediate object code is generated, hence are memory efficient.</td>
<td>Generates intermediate object code which further requires linking, hence requires more memory.</td>
</tr>
<tr>
<td>Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.</td>
<td>It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.</td>
</tr>
</tbody>
</table>
Third Generation Languages (High level languages)

Languages which are somewhere between machine language and the human language.

**FORTRAN** (Formula Translation) – 1957
Language to allow scientists and engineers to program computers. (Early versions – Code did not work on other computers.)

**COBOL** (Common Business Oriented Language) - 1959
Language primarily designed for US government and defense contractors to program business applications on the computer. Grace Hopper was one of the developers of COBOL.
Allowed for any computer to compile the COBOL source code – Write once, run anywhere

**BASIC** (Beginner's All-purpose Symbolic Code) - 1960's
Alternative language to FORTRAN for beginning programming students.
**Third Generation Languages** (High level languages)

**Pascal** (named after Blaise Pascal, 17th century French mathematician) - 1970's

Language to teach proper structured programming.

**Structured programming** = Programming technique used to make programming more productive and easier to write. Stresses simplistic, modular programs.

**ADA** (named after Ada Lovelace (programmed the 19th century 'analytical engine') - late 1970's

Language developed to replace COBOL.
Third Generation Languages  
(High level languages)

**C** (successor to BCPL or "B") - 1970's  
Popular programming language on computers from microcomputers to super computers.  
Faster and more efficient language. Very powerful language.

**Source code example of a C Program** (Displays Hello World! on the screen.)
```c
#include <stdio.h>
main()
{
    printf("Hello World!");
}
```

**C++** (pronounced "C plus plus") - 1980's  
Object oriented language which is compatible with C.

**JAVA**
• Python is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language,
Third Generation Languages (High level languages)

Advantages

- Easier to read, write and debug
- Faster creation of programs

Disadvantages

- Still not a tool for the average user to create software programs
- Requires very good knowledge of programming and of the language
Op-code    Operand   Description
1          LOAD reg. R from cell XY.
2          LOAD reg. R with XY.
3          STORE reg. R at XY.
4          MOVE R to S.
5          ADD S and T into R. (2’s comp.)
6          ADD S and T into R. (floating pt.)
7          OR S and T into R.
8          AND S and T into R.
9          XOR S and T into R.
A          ROTATE reg. R X times.
B          JUMP to XY if R = reg. 0.
C          HALT.

- Adding two numbers together in Assembly
• Compiler takes care of memory locations, registers, etc.