CS101
Introduction to Computer Science

Welcome to CS101 and what I hope is an interesting introduction to the fascinating world of Computer Science (CS). This course serves as an overview of the different topics that are studied throughout the field of CS. We will also introduce you to the Java programming language, which will allow you to write some small programs. The purpose of this course is to give you an idea of what CS is all about and provide a launching point for you to take the next CS course, CS201, Programming Concepts.

Recently there have been many misconceptions about what computer science is. Before we define computer science, let's examine some of them:

Misconception 0: I can put together my own PC, am good with Windows, and can surf the net with ease. These are foundations of computer science.

Not so – computer science studies the theory underlying how the PC works, how an operating system like Windows is designed, and how communication protocols operate on the Internet. Technical competency and computer literacy will help you study these things, but CS foundations are more mathematical and logic-based. There are several successful computer scientists who don't have a clue how to put together their own PC from off-the-shelf parts. If you are interested in a career that focuses on these technical issues, such as PC support, system administration, or network administration, a two-year degree or certificates from the Community and Technical College would be more appropriate.

Misconception 1: Computer science is the study of computers

This definition is too general – while CS does study the computer itself, it is also about how computers can be used.

Misconception 2: Computer science is the study of how to write computer programs

This definition is also incomplete. Computer programs are a large part of CS, but there is much more than the act of coding software. For example, we will spend considerable time developing algorithms. An algorithm is a sequence of steps to solve a problem (e.g., consider a cooking recipe). Before we can write code, we must design an algorithm. The theoretical design and analysis of algorithms is crucial to the construction of an efficient program.

Misconception 3: Computer science is the study of the uses and applications of computers and software.

This definition is more closely aligned with the field of MIS, or Management Information Systems. It studies how computer technology may be used in an organizational setting (e.g., to help a business). In contrast, CS examines the science behind how the computer

itself works and how the software works more so than the application of the technology. If this area interests you, you may wish to pursue a MIS degree from the College of Business.

Now that we've seen what computer science is not, here is the book's definition of computer science:

Computer science is the study of algorithms, including

1. **Their formal and mathematical properties**
2. **Their hardware realizations**
3. **Their linguistic realizations**
4. **Their applications**

Item 1 refers to the correctness and efficiency of an algorithm. For example, what are the steps to take to sort a list of names? How about to play a game of blackjack? Or a game of chess?

Item 2 refers to building the computer itself. What are the hardware components of the computer? How is data represented, for example storing numbers, text, or a picture? Here we will examine the structure of the computer, the central processing unit, and look at various data and number formats.

Item 3 refers to programming languages that implement an algorithm and are executed by the hardware. We'll study programming with Java. Just as a natural language like English has linguistic properties (e.g. semantics, a grammar) we have similar properties for programming languages.

Item 4 refers to the problems that software is used to solve. We'll look at a few examples in networking, artificial intelligence, and data management.

Algorithms

Since computer science is the study of algorithms, let's examine algorithms in more detail. Informally an algorithm is an ordered sequence of instructions to solve a problem:

```
Step 1:     Do stuff
Step 2:     Do stuff
…
Step N:     Stop, problem complete
```

Each operation for "Do Stuff" falls in one of three categories:

1. Sequential operation. This carries out a single well-defined task. For a cooking recipe, this might be "Add 2 eggs" or "Mix ingredients." For a computer program this could be "set background color to gray."

2. Conditional operation. These are the decision-making or "if-then" rules of the algorithm that execute different operations based on a condition. For a cooking recipe, this might be "If no longer soft in the middle, remove from oven; otherwise continue baking.". For a computer program it might be, "If input=3 then exit the program."

3. Iterative operation. These are looping instructions. For a cooking recipe, it might be "Repeat steps 2-4 until mixture is thickened." For a computer program it might be "Repeat steps 1-5 until user input = 3".

Here are some examples of algorithms:

Filling your car with gas:

Step 1: If gas cap is on the left side of the car Then pull up to island on left. Otherwise, pull up to island on right.
Step 2: Release gas lid
Step 3: Open gas cap
Step 4: Slide credit card
Step 5: If authorized, go to step 6. Otherwise, repeat step 5 (i.e. wait)
Step 6: Lift nozzle
Step 7: Push button
Step 8: Insert nozzle into tank
Step 9: Depress handle
Step 10: If gas doesn't stop flowing, repeat step 9. Otherwise, continue.
Step 11: Replace nozzle to pump.
Step 12: Close gas cap.
Step 13: Close gas lid.

Returning the fewest number of coins as change using quarters, dimes, nickels, pennies:
Given: *n* is the value to return as change (1 to 99).
Step 1: Return as many quarters as possible.
    Step 1.1: Set *q* to *n / 25*, discarding any remainder
    Step 1.2: If *q > 0* then return *q* quarters and set *n* to the remainder of *n /25*. (The remainder is the modulus; i.e. *n* is set to *n* mod 25).
Step 2: Return as many dimes as possible.
    Step 2.1: Set *d* to *n / 10* , discarding any remainder
    Step 2.2: If *d* > 0 then return *d* dimes and set *n* to *n* mod 10.
Step 3: Return as many nickels as possible.
    Step 3.1: Set *nick* to *n / 10* , discarding any remainder
    Step 3.2: If *nick* > 0 then return *nick* dimes and set *n* to *n* mod 10.
Step 4: Return *n* pennies.

In this example we are using variables of q, n, d, and nick. The variables are placeholders to store different values of interest. In this case, we store the number of different coins to return as change.

We have also broken up abstract steps into more specific, or more **primitive**, steps (e.g. steps 1.1 and 1.2 are more primitive than Step 1). Ultimately the computer must be told the primitive steps to write a program, but at times it will be sufficient to stop at the more general steps if it is obvious how to implement it.

Running through our algorithm with n = 97:

Step 1.1: q = 97 / 25 = 3, discarding the remainder of 22
Step 1.2: Return 3 quarters, n = 97 mod 25 = 22
Step 2.1: q = 22 / 10 = 2, discarding the remainder of 2
Step 2.2: Return 2 dimes, n = 22 mod 10 = 2
Step 3.1: d = 2 / 5 = 0, discarding the remainder of 2
Step 3.2: Return no nickels
Step 4: Return 2 pennies

Testing if a positive number n is a prime number
A number is prime if it is divisible only by itself and the number 1. Our strategy is to test all numbers between 1 and n-1 and if n is prime, none of these should divide into n.

Step 1: If n equals 1, then n is prime. Stop. Otherwise, continue.
Step 2: Set x to 2
Step 3: If n mod x equals 0 (i.e. n/x has no remainder) then n is not prime. Stop. Otherwise, continue.
Step 4: Set x to x+1
Step 5: If x < n then repeat at step 3. Otherwise, continue.
Step 6: n is prime. If we get here, we've tested all numbers from 2 to n-1 and none of them divided into n Stop.

Running through our algorithm with n = 5:

Step 1: 5 not equal to 1
Step 2: x = 2
Step 3: 5 mod 2 = 1, continue.
Step 4: x = 3
Step 5: 3 < 5, goto step 3
Step 3: 5 mod 3 = 2, continue
Step 4: x = 4
Step 5: 4 < 5, goto step 3
Step 3: 5 mod 4 = 1, continue
Step 4: x = 5
Step 5: 5 is not < 5, continue
Step 6: 5 is prime

Finding all prime numbers between 1 and 100

To do this, we can write an algorithm that uses our algorithm that tests if a number is prime. This is a fairly common practice, so that we can build up more complex algorithms based on existing algorithms.

Step 1: Set n to 2.
Step 2: Check if n is prime using our previous algorithm. If so, output n.
Step 3: Set n to n + 1
Step 4: If n equals 100 then stop. Otherwise, go back to step 2.

Now that we've seen some algorithms, we can talk about a few formalities. An algorithm must be **effectively computable**, meaning that there exists a well-defined computational process that produces a result. For example, if we were to write a program to play chess, we must explicitly tell the program what is a "good" move (e.g., control the center, have more pieces, etc.) We can't leave this as a vague fuzzy notion to be filled in with intuition.

An algorithm must also halt within a finite amount of time, meaning we can't require the process to run forever to find a solution. Some algorithms contain **infinite loops** which never halt (e.g. Step 1: Goto step 1.).