Topic 1 – Introduction and Algorithms

THE BIG PICTURE

Why Programming?

Why should you learn about programming and computer science?

- Programming helps develop problem-solving skills, in particular, the ability to deal with complexity.
- Computer technology is pervasive. We live in the so-called “Age of Information”. Moreover, the impact of the computer on society is increasing.
- Programming skills can be applied in other areas.
- Most “programming” is done by non-computer scientists. Customizing of tools is possible by those who understand programming. This can range from programming complicated queries on databases to running complex simulations with spreadsheets.
- It's a fascinating challenge to teach the computer how to solve hard problems.

History of programming

- The original role of computers and programming was to facilitate ‘number crunching’.
- However, perhaps the early computing pioneers failed to appreciate was the flexibility of numbers.
- The very name *computer* suggests that they thought the only applications for computers would be processing numbers for the purpose of science and engineering.
- They did not recognize or appreciate the fact that numbers can be used to encode just about any information one might want to process.
- There are many examples where numbers are used to represent information but the numeric values of the numbers used have little significance and are not used for actual computation:
    - Post codes,
    - Tax File numbers,
    - Drivers license numbers, ...
- By associating a numeric value with each letter of the alphabet (e.g., a=1, b=2, etc.) we can construct a numerical encoding of any textual information.
- When a computer manipulates numbers of this sort, it is really being used as a symbol or *information processor* rather than a “computer”.

A Universal Information Processor

- It is the fact that just about any information can be encoded using numbers that gives the computer the ability to be a *universal information processor*.

- As a result, the encoding of information forms an important part of all of computer science.

- Recognizing the power of numbers as symbols isn't quite enough to explain the amazing impact computers have had.
    - Simple, hand-held calculators can process numbers too, but they have not had the impact of computers.

- The computer's other fundamental capability – *the ability to follow pre-supplied instruction or programs* – provides the rest of the explanation.

- The ability to change programs makes computers flexible:
    - *They can be adapted to new tasks without being rebuilt.*

- The ability to follow programs makes it possible to exploit the speed of a computer.
    - If the instructions could not be pre-supplied, the computer would spend most of its time waiting for a slow human to press the next control button.

Computer Science Definition

- Computer science provides the theoretical basis for most programming techniques.

- So what is “computer science”, anyway?

- One simple definition that is sometimes used is:

Computer Science is the study of algorithms

- Specifically this entails:

1. Their formal and mathematical properties
    - Studying the behaviour of algorithms to determine whether they are correct and efficient.

2. Their hardware realizations
    - Designing and building computer systems that are able to execute algorithms.

3. Their linguistic realizations
    - Designing programming languages and translating algorithms into these languages so that they can be executed by the hardware.

4. Their applications
    - Identifying important problems and designing correct and efficient software packages to solve these problems

So what exactly are algorithms then?

ALGORITHMS

Introduction

- Our main focus in this unit will be learning how to prepare the instructions needed to ensure that a computer can perform a particular task.

- The instructions presented to a computer are significantly different from the sorts of instructions we might prepare for another human being.

- Computers have no common sense or intuition:
  - They can only perform a task if the instructions provided are accurate and totally unambiguous.

- It must be possible to follow the instructions *without any sense or understanding of their actual purpose*.

  These instructions are called *algorithms.*

- The instructions in an algorithm must specify what to do rather than what can or might be done.
  - Thus, the "instructions" for a board game or a card game would not be considered an algorithm.

- The instructions found in a cook book recipe are much more similar to the instructions that must be included in an algorithm.

Following instructions: People vs Computers

- Our concern in this unit will primarily be to learn how to write good algorithms (i.e. how to write good instructions).

- To appreciate why it might take as long as a semester to accomplish this, consider:
  - Humans, in general are very good at writing bad instructions.
    - Have you ever tried to assemble something complex by following the manufacturer's directions?

- However, luckily humans are generally very good at following bad instructions.
  - Example: A recipe whose first step called for preheating the oven and whose second step called for refrigerating a mixture of ingredients overnight. How many people would be silly enough to leave the oven on overnight?
  - Another example is the shampoo bottle instructions – see later.

- Computers are essentially perfect at following instructions exactly.
  - A computer would definitely leave the oven on overnight.

Formal Definition of an Algorithm

- *Informally*, an algorithm is an ordered sequence of instructions that is *guaranteed* to solve a specific problem.

- Algorithms are important because if you can specify a working algorithm for a problem then you can:
    - Solve the problem
    - Get a computer to solve any equivalent forms of that problem automatically for you.

A formal definition of an algorithm is:

> An Algorithm is a well-ordered collection of unambiguous and effectively computable operations that, when executed, **produces a result** and halts in a finite amount of time.

- The key properties of an algorithm according to this definition are highlighted and we will now go through them.

"...well-ordered..."

- The order of the operations that makes up an algorithm must be correct (i.e., steps cannot be in an order which produces the wrong result.)
- There is also often an efficiency aspect to the order of the instructions – that is, one particular order is more efficient than other orders even if they give the same correct result.
- It is also important that the first operation is clearly indicated so that it is clear where the algorithm starts.
- Similarly its end must be clear, otherwise the algorithm might carry on forever.

"*...unambiguous...*"

- So each instruction must also be clear in what it is that it does

  E.g. Do part 1 or part 2

- The following is less ambiguous but is it completely unambiguous?

  if you are over 18 years of age do Part 1
  else do Part 2

- The following "Shampoo bottle" example is clearly ambiguous:

  STEP 1 Wet hair
  STEP 2 Lather
  STEP 3 Rinse
  STEP 4 Repeat

- Repeat what?
  - Steps 1 – 4? (This will go on forever.)
  - Step 4? (Ditto but will be even less productive!)
  - Steps 1 – 3? (This involves inefficiency because the second time, the hair is already wet!)
  - Steps 2 – 3? (This is correct but it takes human intuition to realise it!)

"...effectively computable operations..."

- The operation must be within the capabilities of the computer it will be executed on.
- For example, a = b + c is effectively computable only if the computer can perform addition.
- There are also external limits on the sorts of computations that can be performed.
- For example, the expression, a = b / c is only effectively computable when c is not zero.

"...halts in a finite amount of time..."

- The algorithm must be capable of finding a result, that is completing within a finite amount of time.
  - Some algorithms may take a very long while to complete for certain sets of data (hours, days, years, centuries...) but they must ultimately be capable of producing a result.
  - Repeating Step 4 ("Repeat") on the shampoo bottle would represent what's called an "infinite loop" and so the algorithm would not halt in a finite amount of time.

Algorithms and Programming Languages

- Learning to write good algorithms is a skill.
- While we can provide advice on how to go about it, there are no set rules that will always enable you to produce a good algorithm for a given task.

- *Practice is really the only way to develop the needed skills.*

- At the same time that you are learning this skill, we will also be looking at programming languages you can use to communicate your algorithms to a computer.
  - In this unit we will be using a language called C.

- However, ultimately the algorithms you write must potentially be able to be implemented in a variety of different languages.
- This applies even to languages you don't yet know!

WHAT CAN ALGORITHMS DO?

There are only three things that an algorithm can do:

- Sequence
- Selection
- Iteration

All algorithms are made up of combinations of these three *structures* and this is known as *structure theorem*.

Sequence

- Sequence relates to steps that are executed one after the other with no variation in their order:
    - Take keys out of pocket.
    - Find correct key.
    - Insert key into door lock.
    - Turn key 90 degrees anti-clockwise.
    - Remove key from lock.
    - Return keys to pocket.
    - Open unlocked door.

- In the sequential algorithms that we write for computers the steps themselves can be of two different types.

- *Computation* steps involve performing some mathematical or computational operation and then doing something with the result of that computation.

- *Input/Output* steps involve either the getting of new data for the program from some source external source (e.g., the user typing at the keyboard) or the outputting of data from the program (e.g., printing it to the screen).

Selection

- These involve making choices depending on some condition.
- These conditions always relate to the truth or falsity of some expression.

- If the condition is true then the algorithm will perform a set of *sequential* statements (as above).
- If the condition is false then it will perform a different set of sequential statements.

```
if grade is greater than or equal to 50 then
        result = pass
else
        result = fail
```

- Selection is important because it allows the algorithm to behave in different ways depending on these conditions.
- In the example above the algorithm will give different results depending on whether the grade is a pass or fail.
- Note that the “result = pass” and “result = fail” are sequential steps involving the assignment of a value or result.

Iteration

- These involve the repetition of certain steps, again according to the truth or falsity of some condition.
- Iterative statements are very important since they take advantage of the speed of computers and the fact that they don't "get bored".
- Computers can perform a boring and repetitive task over and over again, very quickly.
- For example, a trivial searching algorithm might involve iterating through a large collection of data very quickly.
- A person manually searching through that data would get bored, plus they would do it much more slowly.
- If the collection of data is very big then it may not even be feasible for a person to do it manually!

```
item = get first item
while item does not equal search_item AND items left
        item = get next item

if item = search_item
        print "Item found!"
else
        print "Item couldn't be found."
```

- The above is a simple search algorithm.
- It gets the first item of data in some collection and then iterates or "loops" until it finds the item being searched for.
- Once the loop has stopped then either the item has been found or the search simply ran out of items.
- Note the use of sequence, selection and iteration required to form the complete algorithm.

ALGORITHMIC REPRESENTATION

- Programmers use different methods of writing down the designs for their algorithms before they translate into the code for a particular language.

- Although the rules for these representations aren't usually strictly defined, they ***must*** be consistent within themselves.

- It is important when you write algorithms that you use some of these representations and do so consistently.

- One technique is the use of *flow charts* which lay out the flow of control through the program with different shapes to indicate input, calculations, decisions and output.
- We will look at some examples of this later on.

Structured English

Others prefer a *structured English* approach in which the notes describing how a program will work are organised into indented lines that begin to resemble a program...

Here is a rudimentary algorithm for a "menu" program:

```
display menu
read keyboard input
depending on user selection do one of the
following:
    output the 'option 1' message (option 1)
    output the 'option 2' message (option 2)
    output the 'option 3' message (option 3)
```

- There are no real rules with structured English so long as it is consistent and unambiguous.
- In some ways structured English is more suited to high level algorithms and you will see this a lot in the examples for this unit.

Pseudocode

- An alternative to structured English is *pseudocode* which is written very much like a programming language with minimal syntactic conventions.

- Pseudocode tends to be a little more precise and terse than structured English which makes it a preferred technique for writing low-level algorithms.

- **Particularly early on in the unit, pseudocode is definitely the recommended technique for you to use when writing algorithms.**

- The rules for pseudocode are rather loose, since it is to be interpreted by humans, not computers.
- However, again consistency is important.

Example.

```
print "you have 3 choices"
print "enter a for option 1"
print "enter b for option 2"
print "enter c for option 3"

read response

if response == 'a'
    print "You have selected option 1"
if response == 'b'
    print "You have selected option 2"
if response == 'c'
    print "You have selected option 3"
```

- Pseudocode is better than structured English when you first begin to write algorithms because it makes it harder to write ambiguous algorithms.
  - However the exact style you use is up to you.

- The textbook uses several different styles (see the reading schedule on the web site for notes about this) and these are mostly fine.
- The style used predominantly in these lecture notes is also fine.
- Just make sure whatever style you pick, you try to stick with it – particularly within a single algorithm!

DEVELOPING AN ALGORITHMIC SOLUTION

Problem Definition

- Often people take a brief look at a problem and immediately work toward a solution.
  - Unfortunately for most non-trivial problems this approach is not likely to be successful.

- The Golden Rule here is to *closely examine the problem* to make sure you understand what is being asked of you or what it is that needs solving.

- If you get this stage wrong then every thing that follows will also be wrong even though the program itself may run without crashing.

Example: You're asked to write a simple GST calculator.

- What questions should you ask?
  - Is this to take GST exclusive value and add the GST or is it the other way around: a GST inclusive value which you need to output the sell price and GST component?

Other lower-level questions:

- Where will the inputs come from?
  - Keyboard? Database? File? Network resource? Another external device?
- Do the results need to be output in any particular form (i.e., printed on an invoice)?

High-level Algorithm

- An algorithm is a series of steps by which a computer can undertake the required processing to produce the required output.
  - However, for many advanced problems it is too difficult to simply go straight from the problem statement to a complete solution (algorithm).
  - A more gradual approach is often called for.

- A good place to start is to write the high-level steps in the algorithm.
- There are no methods used at this level i.e., no arithmetic operations.
- In other words you simply the major steps to be performed without giving details specifying *how*.

- However, your steps must still “add up”.
- In other words, you should be able to take each of the high level steps you write and expand it into a collection of more detailed steps ultimately representing a complete solution to the problem.
- If you have to add in extra big steps that aren't part of the high level algorithm then this suggests your high level algorithm was incomplete.

  - It is not always necessary to write a high-level algorithm.
  - For example, in the first few topics of this unit the problems we deal with are relatively simple and so a high-level algorithm is not strictly necessary.
  - However, if you think that writing one might help you solve the problem then feel free to do so.

GST Example Cont.

- Get cost of product excluding GST.
- Calculate total cost including GST.
- Output the value to the user.

- Notice the three steps above
    - Input
    - Process
    - Output

You will often see these three phases in simple algorithms.

Define Inputs & Outputs

- What form will the data be entered in?
    - Data types (whole numbers, fractional numbers, single characters, "strings" of characters together).
    - Values like dollar and cents amounts could be read as a single fractional number or as a pair of whole (integer) numbers each representing the dollar and cents amounts.
    - Data obtained from the keyboard, from a file, from a database, across the network?

- How should the output be displayed?
    - Does the data type require special formatting to output correctly?
        - Adding in a dollar sign at the front and printing two decimal places for currency values.
        - Appending a percent sign for percentages.
    - Where should the output go? (To the screen, a file, a database, across the network...)

Create a Test Data Set

- Here we need to consider what results our program should be able to provide.
- For most problems this involves being able to do some calculations to verify the sorts of results the program should obtain for certain data.
- Note that this implicitly requires a basic understanding of how to solve the problem but for nearly all problems this is not an issue.

Example Cont.

- We need to multiply the ex-GST price by 1.1 to calculate the new price including GST.
- Even though this is a critical part of the solution, it is a simple calculation to make.
- This is important because it allows us to do test calculations to see what sort of results the program should produce.
- This process verifies our algorithm as correct and is called *desk checking*.
- The following table represents some possible values that you can test in a preliminary way to understand the behaviour of the algorithm.
- Note that these are not necessarily ideal test values!

Ex-GST Price	Expected Result Inclusive GST Price
0	0
10	11
25	27.5
20000	22000
-5	-5.5
21.95	24.145

- Also, the ones highlighted in red are those that reveal potential issues which would need to be dealt with to provide a complete and reliable algorithm.

Create a Low-level Algorithm

- Here we put the individual steps that a computer will require to process our input to the desired output.

- Here we take our high-level algorithm and break each step down, or *decompose* it.

- This method is called *top-down design*.
- It allows us to look at the big picture first then break the problem up into smaller and smaller parts.
- For complex and difficult problems this makes producing a complete solution much easier.
- Note this example illustrates the process but doesn't reflect how you should represent each step.

Example Cont.

- Get cost of product excluding GST.
  - Print "Enter price of product ex-GST: "
  - Read user input into *price*

- Calculate total cost including GST.
  - if *price* < 0 then print "Invalid input!" and exit
  - *total = price * 1.1*

- Output the value to the user.
  - *total = total* rounded to 2 decimal places
  - print "Total price is: $", *total*

Code the Algorithm

- It is at this point where we take a programming language, capable of carrying out our desired task and use it to *implement* our algorithm.

- The major task here is to convert the algorithm into the correct *syntax* of the chosen language.

- The syntax of a language is the collection of rules for how a particular operation is expressed and written in that language.

- There are often many syntactical similarities between languages, for example C, C++ and Java are very similar in many ways.

- So even though we will cover only C in this unit, by learning it you will find it very easy to learn the syntax of other languages.

We will deal with the basic components of a program in a later topic.

Apply the Test Data

- Now we will take the test data set that we created earlier in this process and input it into our program and check that it gives the expected results.
- This allows us to confirm that our implemented program matches the algorithm we designed (and desk-checked) previously.

- If it does produce expected results then you know that your program matches your algorithm although it may not be completely bug free.

- Test data sets should be created with great care: if the test data does not properly and comprehensively test the correctness of the algorithm then there may be significant bugs that remain undiscovered.

- Selection of test data is therefore critical!
- For this reason, a significant amount of marks are allocated to it in the assignments for this unit!
- How to correctly select test data will be discussed later in the unit.

SUMMARY

- Computers are powerful devices because they can not just perform computations but also because of their ability to encode information into numbers and to follow specified instructions.

- Computer science may be described as the study of algorithms.
- These are ordered sequences of instructions to solve a given problem.
- Computer scientists develop algorithms to solve problems and then convert these into programs able to be run on a computer.

- Algorithms can be made up of sequential steps, selective structures and iterative structures.
- They can be represented in the form of structured English and pseudocode.

- When developing an algorithm it is often best to start with the general or “high-level” steps then refine each of these by specifying the lower-level individual steps that make these up.

- After coding it is important to apply test data to the program to ensure that it works correctly.