Introduction to Programming

9: Developing an Algorithm

Algorithms – Specification and Design

- Developing an algorithm is more than writing code
  - May not yet have decided on a language
  - Code does not document the process well
  - Some algorithms are better described by diagrams, flowcharts etc.
  - Constraints and assumptions should be documented
- Typically, we would use some form of specification language or notation to describe an algorithm
  - Several available, suited to various types of task or application domain
  - Need something between a plain English description (ambiguous, wordy, difficult to add precise detail to) and program code (too prescriptive, limiting, not easy to read)

Methods

- Rigorous development is necessary to maintain
  - Communication
  - Development style
  - Control of complexity
  - Verification and validation
- We can use a number of methods/tools
  - Program Design Language (structured English, Pseudocode)
  - Flowcharts
  - Data designs and data flow diagrams
  - A formal methodology or notation (e.g. UML, Z, VDM)

Steps...

- Formal/precise statement of requirements
- Statement of prerequisites
  - Inputs
  - Function Requirements
  - Outputs
  - Assumptions
- Specifications
  - Precise descriptions of the above
- Nomenclature
  - Good naming style of items in design/code
- Testing
  - BEFORE we build a routine, need to specify tests that will be adequate to demonstrate its suitability

Algorithms

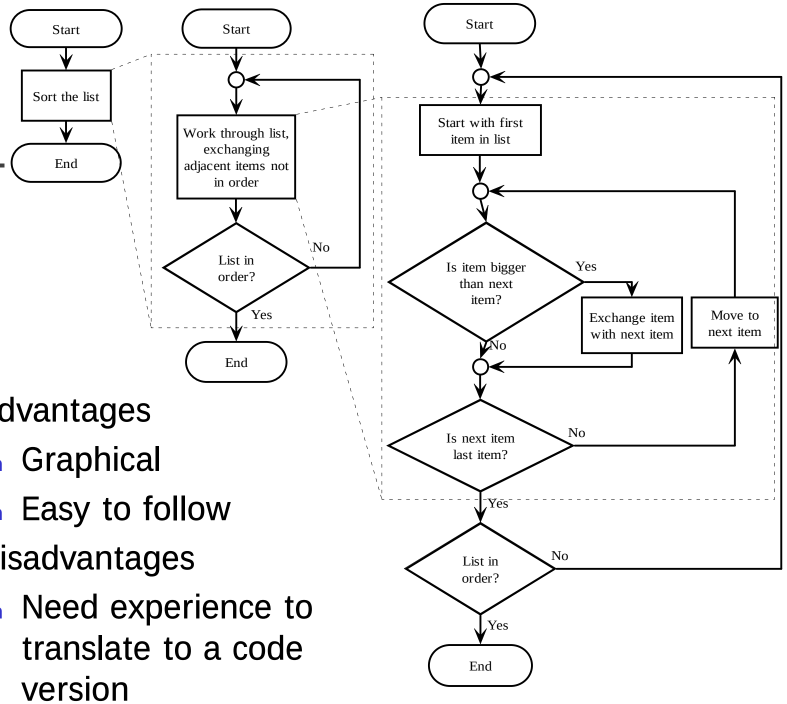
- Few people ***invent*** an algorithm
  - Those that do generally get recognition for this
- Most algorithms come from
  - Translation of current (manual) procedures
  - Text books (e.g. the Art of Programming)
  - Experience and received wisdom

Example Algorithm

- Requirements
  - Sort a list of data items (e.g. strings) into a specific order (e.g. alphabetical)
- Several standard algorithms for this exist
  - Different versions are faster, occupy less memory as they operate, or are easier to code
- e.g. Bubblesort
  - Slowest, but one of the easiest to code
  - Brief description: Go through the list from beginning to end, exchanging adjacent items if they are not in order – repeat process until list is sorted

Formalizing the algorithm

- Indicate the form of information to be processed (a list of strings)
- Define the operation in detail, either grammatically (PDL) or diagrammatically (Flowchart)
  - Use top-down design (so it is possible to start with a general description and add detail until it is fully developed)
- Stop developing the design when there is sufficient detail to allow it to be converted into code
  - Usually on a line for line basis (structured English)
  - Usually when each operation can be implemented in a line of code (flowchart)
- Examples of both approaches...


Start
Sort the list
End
Start
Work through list, exchanging adjacent items not in order
List in order?
No
Yes
End
Start
Start with first item in list
Is item bigger than next item?
Yes
Exchange item with next item
No
Move to next item
Is next item last item?
No
Yes
List in order?
No
Yes
End

- Advantages
  - Graphical
  - Easy to follow
- Disadvantages
  - Need experience to translate to a code version

PDL Style

```
Do
   Clear Exchange-Flag
   For Item = First to Second-Last
      If Item > Next-Item
         Exchange Item and Next-Item
         Set Exchange-flag
      End If
   Next
Loop Until Exchange-Flag is clear
```

Note – these are needed to find out if the list is sorted.

- Advantages
  - Close to code
  - Indicates structures to use
- Disadvantages
  - Need to understand programming concepts

The algorithm isn’t everything

- Need indication of inputs/outputs
- Need indication of in-scope data changed
- Additional (non-algorithmic) detail
  - Name of developer
  - Date/Version information
  - Indication of target of code (application, library)

Coding

- By this stage, should be easy
- For Java development, operational considerations
  - Incorporate routine into main code body (not usually a good idea)
  - Package code up as a static method or subroutine (possibly in its own class)
    - Method specification – name, parameters, return type, assumptions and effects
    - Method body - define the workings of the code

e.g. Java specification

javadoc comments, used to generate HTML documentation from the code (see pages 146 to 148 of the textbook) à

```
/**
 *   Sorts the String elements from list[0] to
 *   list[size-1] in to ascending order.
 *
 *  @param list, the list to be sorted
 *  @param size, the number of elements in
 *   the list that are to be sorted
 *   (list[0] to list[size-1])
 *  @throw NullPointerException if any of the
 *   String elements to be sorted , or the
 *   list itself are null
 */
public static void bubbleSort(String[] list,
                                int size) { … }
```

Notes on the specification

- Describes what the method does, using a standard Java format for comments that can be used to generate documentation
- The method header had two parameters – one for the array of String and one for the number of elements to sort
  - Could just have one parameter, the list, and use **list.length** instead of size, but this requires that all the array elements refer to Strings (the array may not be full and the last few elements might be **null**)
    - It is more flexible to allow the array to contain some null elements at the end of the array, but we did not have to do this
  - Algorithm does assume that none of the Strings at locations list[0] to list[size-1] are **null**

Comparing Strings

- To sort a list of strings, need to compare the strings for order
- For primitive numeric types in Java can use the relational operators <, <=, >, >=
- No relational operators for classes in Java

The compareTo() method

- Classes that have a natural ordering define a method named compareTo() that can be called to compare the ordering of two instances of the class
- For String:

```
public int compareTo(String that);
```

- Example of usage (note lexicographic ordering):

```
String s = "cat";
int comp = s.compareTo("dog"); // -ve, less
comp = s.compareTo("cat");     // 0, equal
comp = s.compareTo("canine");  // +ve, more
comp = s.compareTo("CAT");     // +ve, more
```

More on compareTo()

- The method returns a negative number if the value of the argument is larger, zero if the value of the argument is equal to, or a positive number if the value of the argument is smaller than the value of the object whose compareTo() method is called
- Use the tests

  (a.compareTo(b) > 0) for a greater than b

  (a.compareTo(b) < 0) for a less than b

  (a.compareTo(b) == 0) for a equal to b

  - A common convention is to assign result of call to compareTo() to an **int** variable, comp, whose value is then compared with 0, as in next slide

The Implementation

The code (in a Java class) à

```
public static void bubbleSort(String[] list, int size)
{
   int pos;
   int comp;
   String temp;
   boolean exchangeFlag;
   do {
       exchangeFlag = false;
       for (pos=0; pos<(size-1); pos++) {
         comp = list[pos].compareTo(list[pos+1]);
         if (comp > 0) {
            temp = list[pos];
            list[pos] = list[pos+1];
            list[pos+1] = temp;
            exchangeFlag = true;
         }
      }
   } while(exchangeFlag);
}
```

Testing an implementation

Using main() to test a subroutine, simplest to put main() in the same class as the method(s) being tested.

```
public static void main( String[] args) {
   String[] stringList = new String[20];
   // Set up test data…
   stringList[0] = "Test";
   stringList[1] = "Data";
   ...
   stringList[9] = "Alphabetical";

   // Call the subroutine…
   bubbleSort( stringList, 10 );

   // check results… e.g. visually
   for (int i=0; i<10; i++)
      System.out.println(stringList[i]);
}
```

Working on Algorithms – Key points

- Apart from your time at university, you will seldom get the chance to work on a project on your own
  - Work you do must be
    - Done to an accepted standard
    - Readable
    - Fixable
    - Able to integrate with other work
- The only accepted approach to these goals is to write code as part of a rigorous development process
  - Employers will not accept anything else