MT4113, Computing in Statistics

Lecture 1 - Hardware, software and algorithms

17 September 2018

Computers (from a statistician's persepctive)

What is a computer?

- Approximate definition:

  A device that accepts information (in the form of digitized data) and manipulates it according to a sequence of instructions to produce output.

What is a computer?

- Approximate definition:

  A device that accepts information (in the form of digitized data) and manipulates it according to a sequence of instructions to produce output.

- We can divide the device into two parts:

What is a computer?

- Approximate definition:

  A device that accepts information (in the form of digitized data) and manipulates it according to a sequence of instructions to produce output.

- We can divide the device into two parts:
  - Hardware - physical computer equipment

What is a computer?

- Approximate definition:

  A device that accepts information (in the form of digitized data) and manipulates it according to a sequence of instructions to produce output.

- We can divide the device into two parts:
  - Hardware - physical computer equipment
  - Software - set of instructions that operate the hardware $\sim$ computer program

Computer architecture

Hardware

- Central Processing Unit (CPU) – controls the computer and executes instructions
  (N.B. Many computers also contain graphics processing units (GPU's), which can be used for numerical computation.)

Computer architecture

Hardware

- Central Processing Unit (CPU) – controls the computer and executes instructions
  (N.B. Many computers also contain graphics processing units (GPU's), which can be used for numerical computation.)
- Memory – fast storage

Computer architecture

Hardware

- Central Processing Unit (CPU) – controls the computer and executes instructions
  (N.B. Many computers also contain graphics processing units (GPU's), which can be used for numerical computation.)
- Memory – fast storage
  - Registers – within the CPU, superfast

Computer architecture

Hardware

- Central Processing Unit (CPU) – controls the computer and executes instructions
  (N.B. Many computers also contain graphics processing units (GPU's), which can be used for numerical computation.)
- Memory – fast storage
  - Registers – within the CPU, superfast
  - Random Access Memory (RAM) – slower

Computer architecture

Hardware

- Central Processing Unit (CPU) – controls the computer and executes instructions
  (N.B. Many computers also contain graphics processing units (GPU's), which can be used for numerical computation.)
- Memory – fast storage
  - Registers – within the CPU, superfast
  - Random Access Memory (RAM) – slower
- Mass storage – e.g., hard disk drives – (traditionally) much slower

Computer architecture

Hardware

- Central Processing Unit (CPU) – controls the computer and executes instructions
  (N.B. Many computers also contain graphics processing units (GPU's), which can be used for numerical computation.)
- Memory – fast storage
  - Registers – within the CPU, superfast
  - Random Access Memory (RAM) – slower
- Mass storage – e.g., hard disk drives – (traditionally) much slower
- Input/Output – keyboard, screen, printer, etc.

Software

- System software – e.g., Microsoft Windows, Linux

Software

- System software – e.g., Microsoft Windows, Linux
- Application software – e.g., Microsoft Word, SAS, R

Software

- System software – e.g., Microsoft Windows, Linux
- Application software – e.g., Microsoft Word, SAS, R
- Programming software – e.g., Microsoft Visual C, GNU C, R

Software

- System software – e.g., Microsoft Windows, Linux
- Application software – e.g., Microsoft Word, SAS, R
- Programming software – e.g., Microsoft Visual C, GNU C, R
  - Implementations of programming languages – C, S, etc.

Programming languages

Classified according to:

- Level of abstraction

Programming languages

Classified according to:

- Level of abstraction
  - low level = close to the specific computer hardware - black-belt stuff

Programming languages

Classified according to:

- Level of abstraction
  - low level = close to the specific computer hardware - black-belt stuff
  - high level = far from the hardware (so can run on many different systems) and closer to natural language - can focus on tasks to be achieved

Programming languages

Classified according to:

- Level of abstraction
  - low level = close to the specific computer hardware - black-belt stuff
  - high level = far from the hardware (so can run on many different systems) and closer to natural language - can focus on tasks to be achieved
- Generality: higher level languages tend to be more specialized in application

Programming languages

Classified according to:

- Level of abstraction
  - low level = close to the specific computer hardware - black-belt stuff
  - high level = far from the hardware (so can run on many different systems) and closer to natural language - can focus on tasks to be achieved
- Generality: higher level languages tend to be more specialized in application
- Speed/Efficiency: lower level is generally faster (although not always true)

Programming languages

Classified according to:

- Level of abstraction
  - low level = close to the specific computer hardware - black-belt stuff
  - high level = far from the hardware (so can run on many different systems) and closer to natural language - can focus on tasks to be achieved
- Generality: higher level languages tend to be more specialized in application
- Speed/Efficiency: lower level is generally faster (although not always true)
- Generations: 1st-5th

Programming language generations

- 1st generation (1GL): machine language - CPU-specific set of instructions. E.g. `10110000 01100001` on a Pentium-type CPU means move the value 97 into a certain register.

Programming language generations

- 1st generation (1GL): machine language - CPU-specific set of instructions. E.g. `10110000 01100001` on a Pentium-type CPU means move the value 97 into a certain register.
  - Relatively few different instructions (≈ 100 in an Intel chip) and many of these are required to achieve anything useful.

Programming language generations

- 1st generation (1GL): machine language - CPU-specific set of instructions. E.g. `10110000 01100001` on a Pentium-type CPU means move the value 97 into a certain register.
  - Relatively few different instructions (≈ 100 in an Intel chip) and many of these are required to achieve anything useful.
- 2nd generation (2GL): assembly language.

Programming language generations

- 1st generation (1GL): machine language - CPU-specific set of instructions. E.g. `10110000 01100001` on a Pentium-type CPU means move the value 97 into a certain register.
  - Relatively few different instructions ($\approx$ 100 in an Intel chip) and many of these are required to achieve anything useful.
- 2nd generation (2GL): assembly language.
  - A human-readable form of machine language. E.g., the above in assembly is `mov al, 061h`

Programming language generations

- 1st generation (1GL): machine language - CPU-specific set of instructions. E.g. `10110000 01100001` on a Pentium-type CPU means move the value 97 into a certain register.
  - Relatively few different instructions ($\approx$ 100 in an Intel chip) and many of these are required to achieve anything useful.
- 2nd generation (2GL): assembly language.
  - A human-readable form of machine language. E.g., the above in assembly is `mov al, 061h`
  - `al` is the name of the register and `61h` is 61 in hexadecimal, which is 97 in decimal

- 3rd generation (3GL): more human-friendly, CPU-independent language with variables, data and code structures.

- 3rd generation (3GL): more human-friendly, CPU-independent language with variables, data and code structures.
  - E.g.
    FORTRAN

    ```
    IF(X.EQ.12) LET A = B
    ```

    BASIC

    ```
    if (x=12) then a = b
    ```

    Pascal

    ```
    if (x=12) then a := b;
    ```

    C

    ```
    if (x==12) a = b
    ```

- 3rd generation (3GL): more human-friendly, CPU-independent language with variables, data and code structures.
  - E.g.
    FORTRAN

    ```
    IF(X.EQ.12) LET A = B
    ```

    BASIC

    ```
    if (x=12) then a = b
    ```

    Pascal

    ```
    if (x=12) then a := b;
    ```

    C

    ```
    if (x==12) a = b
    ```
  - Also *object-oriented* languages such as C++ and Java.

- 3rd generation (3GL): more human-friendly, CPU-independent language with variables, data and code structures.
  - E.g.
    FORTRAN

    ```
    IF(X.EQ.12) LET A = B
    ```

    BASIC

    ```
    if (x=12) then a = b
    ```

    Pascal

    ```
    if (x=12) then a := b;
    ```

    C

    ```
    if (x==12) a = b
    ```

  - Also *object-oriented* languages such as C++ and Java.
  - Most programs you use (e.g., Windows, Unix, R, etc.) are written in a 3GL.

- 4th generation (4GL): language designed with a specific application in mind – lots of built-in capabilities for that application.

- 4th generation (4GL): language designed with a specific application in mind – lots of built-in capabilities for that application.
  - database query languages (SQL), graphical user interface (GUI) creators (Visual Basic), mathematics languages (Mathematica, Maple), statistics languages (R, SAS).

- 4th generation (4GL): language designed with a specific application in mind – lots of built-in capabilities for that application.
  - database query languages (SQL), graphical user interface (GUI) creators (Visual Basic), mathematics languages (Mathematica, Maple), statistics languages (R, SAS).
- 5th generation (5GL): language based around solving problems, given the problem specification. User does not need to explicitly write the algorithm for solving the problem.

- 4th generation (4GL): language designed with a specific application in mind – lots of built-in capabilities for that application.
  - database query languages (SQL), graphical user interface (GUI) creators (Visual Basic), mathematics languages (Mathematica, Maple), statistics languages (R, SAS).
- 5th generation (5GL): language based around solving problems, given the problem specification. User does not need to explicitly write the algorithm for solving the problem.
  - Prolog. Used is in artificial intelligence studies.

Compiled vs Interpreted

- All languages must be turned into machine code before they can be executed by a computer

Compiled vs Interpreted

- All languages must be turned into machine code before they can be executed by a computer
- Two approaches:

Compiled vs Interpreted

- All languages must be turned into machine code before they can be executed by a computer
- Two approaches:
  - Interpret: turn each line into machine code as it is entered using an 'interpreter' and run it straight away.

Compiled vs Interpreted

- All languages must be turned into machine code before they can be executed by a computer
- Two approaches:
  - Interpret: turn each line into machine code as it is entered using an 'interpreter' and run it straight away.
  - Compile: once the code is written and saved into a file, turn it all into machine code in one go using a 'compiler'. Then, it can be run.

Compiled vs Interpreted

- All languages must be turned into machine code before they can be executed by a computer
- Two approaches:
  - Interpret: turn each line into machine code as it is entered using an 'interpreter' and run it straight away.
  - Compile: once the code is written and saved into a file, turn it all into machine code in one go using a 'compiler'. Then, it can be run.
- Pros and cons:

Compiled vs Interpreted

- All languages must be turned into machine code before they can be executed by a computer
- Two approaches:
  - Interpret: turn each line into machine code as it is entered using an 'interpreter' and run it straight away.
  - Compile: once the code is written and saved into a file, turn it all into machine code in one go using a 'compiler'. Then, it can be run.
- Pros and cons:
  - Interpreted code provides instant feedback – good for short, run once jobs. Tend to be used by 4GLs.

Compiled vs Interpreted

- All languages must be turned into machine code before they can be executed by a computer
- Two approaches:
  - Interpret: turn each line into machine code as it is entered using an 'interpreter' and run it straight away.
  - Compile: once the code is written and saved into a file, turn it all into machine code in one go using a 'compiler'. Then, it can be run.
- Pros and cons:
  - Interpreted code provides instant feedback – good for short, run once jobs. Tend to be used by 4GLs.
  - Compiled code runs faster (compiler can *optimize*) – good for jobs that will be run many times. Tend to be used by 3GLs.

(Trivial) example of optimization

- Slower:

```
for i = 1 to 10
  j[i] = log(k) + i
next
```

(Trivial) example of optimization

- Slower:

```
for i = 1 to 10
  j[i] = log(k) + i
next
```

- An optimizing compiler could recognize the inefficiency and automatically turn it into something like the following while compiling.

(Trivial) example of optimization

- Slower:

```
for i = 1 to 10
  j[i] = log(k) + i
next
```

- An optimizing compiler could recognize the inefficiency and automatically turn it into something like the following while compiling.
- Faster:

```
l = log(k)
for i = 1 to 10
  j[i] = l + i
next
```

What programming language/package should we use for statistics?

- For simple *do once* stuff (by the way, *do once* is a myth), we can use our favourite stats package, perhaps via a GUI

What programming language/package should we use for statistics?

- For simple *do once* stuff (by the way, *do once* is a myth), we can use our favourite stats package, perhaps via a GUI
  - but there is no *reproducible trail* of how the analysis was done

What programming language/package should we use for statistics?

- For simple *do once* stuff (by the way, *do once* is a myth), we can use our favourite stats package, perhaps via a GUI
  - but there is no *reproducible trail* of how the analysis was done
- For more complex stuff, but still only *do once* or *do a few times*, use a statistical 4GL within an interpreter

What programming language/package should we use for statistics?

- For simple *do once* stuff (by the way, *do once* is a myth), we can use our favourite stats package, perhaps via a GUI
  - but there is no *reproducible trail* of how the analysis was done
- For more complex stuff, but still only *do once* or *do a few times*, use a statistical 4GL within an interpreter
- For production software, or where efficiency is important

What programming language/package should we use for statistics?

- For simple *do once* stuff (by the way, *do once* is a myth), we can use our favourite stats package, perhaps via a GUI
  - but there is no *reproducible trail* of how the analysis was done
- For more complex stuff, but still only *do once* or *do a few times*, use a statistical 4GL within an interpreter
- For production software, or where efficiency is important
  - use a 3GL and compile the code, or

What programming language/package should we use for statistics?

- For simple *do once* stuff (by the way, *do once* is a myth), we can use our favourite stats package, perhaps via a GUI
  - but there is no *reproducible trail* of how the analysis was done
- For more complex stuff, but still only *do once* or *do a few times*, use a statistical 4GL within an interpreter
- For production software, or where efficiency is important
  - use a 3GL and compile the code, or
  - prototype in a 4GL and then re-write the bits that are slow in a 3GL which you call from the 4GL

Software for statistics

Which software to use for statistics?

Most of the time you have no choice!

- Use what the client uses/wants

Which software to use for statistics?

Most of the time you have no choice!

- Use what the client uses/wants
- Your company only has a license for one piece of software, or your IT department will only support one package

Which software to use for statistics?

Most of the time you have no choice!

- Use what the client uses/wants
- Your company only has a license for one piece of software, or your IT department will only support one package
- You are taking MT4113 and we are only teaching you R

If you do have a choice:

- If you have a zero budget, use R – it's free!

If you do have a choice:

- If you have a zero budget, use R – it's free!
- If you only use the package rarely and want to do something quite standard, choose something easy to use (point and click) SPSS, etc.

If you do have a choice:

- If you have a zero budget, use R – it's free!
- If you only use the package rarely and want to do something quite standard, choose something easy to use (point and click) SPSS, etc.
- For enormous datasets, use SAS (or maybe Microsoft R Open, or R with Hadoop)

If you do have a choice:

- If you have a zero budget, use R – it's free!
- If you only use the package rarely and want to do something quite standard, choose something easy to use (point and click) SPSS, etc.
- For enormous datasets, use SAS (or maybe Microsoft R Open, or R with Hadoop)
- If you are doing research in statistics, start with R (or maybe MATLAB/Mathematica)

- For computer-intensive work, or something that you wish to automate

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)
  - Find out which bits are slow and port them to a 3GL

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)
  - Find out which bits are slow and port them to a 3GL
  - . . . or re-write the whole thing into a 3GL

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)
  - Find out which bits are slow and port them to a 3GL
  - . . . or re-write the whole thing into a 3GL
  - Consider what kind of user interface you need

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)
  - Find out which bits are slow and port them to a 3GL
  - . . . or re-write the whole thing into a 3GL
  - Consider what kind of user interface you need
- For specialized applications, look at the specialized packages, e.g.,

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)
  - Find out which bits are slow and port them to a 3GL
  - ... or re-write the whole thing into a 3GL
  - Consider what kind of user interface you need
- For specialized applications, look at the specialized packages, e.g.,
  - JAGS/BUGS/Stan - Bayesian calculations

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)
  - Find out which bits are slow and port them to a 3GL
  - . . . or re-write the whole thing into a 3GL
  - Consider what kind of user interface you need
- For specialized applications, look at the specialized packages, e.g.,
  - JAGS/BUGS/Stan - Bayesian calculations
  - EpiInfo - epidemeology

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)
  - Find out which bits are slow and port them to a 3GL
  - . . . or re-write the whole thing into a 3GL
  - Consider what kind of user interface you need
- For specialized applications, look at the specialized packages, e.g.,
  - JAGS/BUGS/Stan - Bayesian calculations
  - EpiInfo - epidemeology
  - Distance - widlife surveys (made in St Andrews!)

- For computer-intensive work, or something that you wish to automate
  - Prototype with an easy-to-use but extensible package (R)
  - Find out which bits are slow and port them to a 3GL
  - ... or re-write the whole thing into a 3GL
  - Consider what kind of user interface you need
- For specialized applications, look at the specialized packages, e.g.,
  - JAGS/BUGS/Stan - Bayesian calculations
  - EpiInfo - epidemeology
  - Distance - widlife surveys (made in St Andrews!)
  - ... many many others ...

R

- "A language and environment for statistical computing and graphics"

R

- "A language and environment for statistical computing and graphics"
- Open source, free

R

- "A language and environment for statistical computing and graphics"
- Open source, free
- Widely used by academic statisticians

R

- "A language and environment for statistical computing and graphics"
- Open source, free
- Widely used by academic statisticians
- Based on the S 4GL

R

- "A language and environment for statistical computing and graphics"
- Open source, free
- Widely used by academic statisticians
- Based on the S 4GL
- Extremely extensible

R

- "A language and environment for statistical computing and graphics"
- Open source, free
- Widely used by academic statisticians
- Based on the S 4GL
- Extremely extensible
- Rapidly developing and maturing

- Current version is 3.5.1

- Current version is 3.5.1
- Widely used in academia, making its mark elsewhere

- Current version is 3.5.1
- Widely used in academia, making its mark elsewhere
- Core development team of approx 20 members, with many hundreds contributing extensions (packages)

- Current version is 3.5.1
- Widely used in academia, making its mark elsewhere
- Core development team of approx 20 members, with many hundreds contributing extensions (packages)
- Supported by the R Foundation and (commercial) R Consortium.

- Current version is 3.5.1
- Widely used in academia, making its mark elsewhere
- Core development team of approx 20 members, with many hundreds contributing extensions (packages)
- Supported by the R Foundation and (commercial) R Consortium.
- `useR!` conferences, etc.

Why use R?

- Pros:

Why use R?

- Pros:
  - Contains cutting-edge methods (usually as add-on packages)

Why use R?

- Pros:
  - Contains cutting-edge methods (usually as add-on packages)
  - Highly extensible

Why use R?

- Pros:
  - Contains cutting-edge methods (usually as add-on packages)
  - Highly extensible
  - Cannot beat the price

Why use R?

- Pros:
  - Contains cutting-edge methods (usually as add-on packages)
  - Highly extensible
  - Cannot beat the price
- Cons:

Why use R?

- Pros:
  - Contains cutting-edge methods (usually as add-on packages)
  - Highly extensible
  - Cannot beat the price
- Cons:
  - Steep learning curve

Why use R?

- Pros:
  - Contains cutting-edge methods (usually as add-on packages)
  - Highly extensible
  - Cannot beat the price
- Cons:
  - Steep learning curve
  - Less well supported than a commercial package?

Why use R?

- Pros:
  - Contains cutting-edge methods (usually as add-on packages)
  - Highly extensible
  - Cannot beat the price
- Cons:
  - Steep learning curve
  - Less well supported than a commercial package?
  - Greater tendency to ignore backwards compatibility?

Learning resources for R

- Introduction to Stats with R books (see reading list)

Learning resources for R

- Introduction to Stats with R books (see reading list)
- R project home page http://www.r-project.org/

Learning resources for R

- Introduction to Stats with R books (see reading list)
- R project home page http://www.r-project.org/
  - Up-to-date list of books

Learning resources for R

- Introduction to Stats with R books (see reading list)
- R project home page http://www.r-project.org/
  - Up-to-date list of books
- Other books see additional reading list (on Moodle)

Learning resources for R

- Introduction to Stats with R books (see reading list)
- R project home page http://www.r-project.org/
  - Up-to-date list of books
- Other books see additional reading list (on Moodle)
- R bloggers - a collection of hundreds of blogs on R http://www.r-bloggers.com/

Evolution of an R programmer

- Phase 1

Evolution of an R programmer

- Phase 1
  - R as calculator

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement
  - one or two comments

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement
  - one or two comments
  - not intended for use (or viewing) by others

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement
  - one or two comments
  - not intended for use (or viewing) by others
- Phase 3

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement
  - one or two comments
  - not intended for use (or viewing) by others
- Phase 3
  - modular programming with functions

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement
  - one or two comments
  - not intended for use (or viewing) by others
- Phase 3
  - modular programming with functions
  - undertaking a sufficiently complex analysis that organisation becomes critical

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement
  - one or two comments
  - not intended for use (or viewing) by others
- Phase 3
  - modular programming with functions
  - undertaking a sufficiently complex analysis that organisation becomes critical
  - description of input and output are critical to ensure reusability

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement
  - one or two comments
  - not intended for use (or viewing) by others
- Phase 3
  - modular programming with functions
  - undertaking a sufficiently complex analysis that organisation becomes critical
  - description of input and output are critical to ensure reusability
- Note: when performing functional programming,

Evolution of an R programmer

- Phase 1
  - R as calculator
- Phase 2
  - the *script*
  - 10-15 lines of code strung together to perform a task
  - might have a loop, maybe even an if-statement
  - one or two comments
  - not intended for use (or viewing) by others
- Phase 3
  - modular programming with functions
  - undertaking a sufficiently complex analysis that organisation becomes critical
  - description of input and output are critical to ensure reusability
- Note: when performing functional programming,
  - there are elements of the R language not previously used that become useful

Preview of next lecture

Preview of next lecture

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time

Preview of next lecture

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features

Preview of next lecture

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features
  - An ordered sequence

Preview of next lecture

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features
  - An ordered sequence
  - Unambiguous and well defined instructions each instruction is clear, do-able, and can be done without difficulty

Preview of next lecture

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features
  - An ordered sequence
  - Unambiguous and well defined instructions each instruction is clear, do-able, and can be done without difficulty
  - Performs some task algorithm needs to be complete, with nothing left out

Preview of next lecture

- Algorithm: an *ordered* sequence of *unambiguous* and well-defined instructions for *performing some task* and *halting* in finite time
- Important features
  - An ordered sequence
  - Unambiguous and well defined instructions each instruction is clear, do-able, and can be done without difficulty
  - Performs some task algorithm needs to be complete, with nothing left out
  - Halts in finite time i.e., the algorithm needs to terminate

Description of Assignment 1

Description of Assignment 1

- We will describe the first assignment, which is available from the class Moodle.

Description of Assignment 1

- We will describe the first assignment, which is available from the class Moodle.
- Set: today! Due: 2 weeks today, 1 Oct.

Description of Assignment 1

- We will describe the first assignment, which is available from the class Moodle.
- Set: today! Due: 2 weeks today, 1 Oct.
- Peer review. Set 1 Oct. Due 8 Oct.