B16 Software Engineering

Structured Programming

Lecture 1

Dr Andrea Vedaldi
4 lectures, Michaelmas Term
http://www.vlfeat.org/~vedaldi/teach.html
B16 Software Engineering

Four Courses

- **Structured Programming**
  - Basic principles of software engineering
  - Writing structured code in a procedural language

- **Object Oriented Programming**
  - Understanding the key principles of object-oriented design and programming

- **Computer Communications and Networking**
  - How devices and computers communicate
  - The Internet

- **Operating Systems**
  - The glue between the programmer and controlling the real-world
B16 Software Engineering

Meta Learning Outcomes

• Understand the *importance of good design practice in software*, and the role that **structured** and **object oriented programming** ideas play in this.

• Understand of how to develop applications that can interact with the outside world (*i.e.* programs as part of bigger engineering systems) via the **operating system** and via **inter-computer** and **computer-device communications**

• Understand how ideas in 4\textsuperscript{th} year options and projects can actually be implemented in software.
Software Engineering vs Structured Programming

Not really a course about software engineering...

- **Software engineering**
  - Mostly about concepts/principles of design, modularity, abstraction, encapsulation, etc.

- **Structured programming**
  - Revision, coding in C and MATLAB, control flow, variables, data types
  - The mapping from high-level language to architecture (including memory)

- **Functions**
  - Code re-use, parameters, libraries

- **Data structures**
  - Structures/classes, arrays

- **Algorithms**

- **Recursion**
Learning Outcomes

The course will aim to give a good understanding of basic software design methods, and emphasize the need to produce well-structured maintainable computer software. The course will concentrate on **principles**, but these will be reinforced **with examples in MATLAB and C/C++** programming languages. Specifically, by the end of this course students should:

- Understand concepts of basic program design techniques that can be applied to a variety of programming languages.
- Understand the need for structured programming in software projects.
- Understand the mechanics of function calls and of recursion.
- Understand the role, uses and advantages of compound data structures.
- Be able to recognise, produce and/or maintain well structured programs.
Texts

- Lipmann and Lajoie, *C++ Primer*, Addison-Wesley, 2005
- Goodrich et al., *Data structures and algorithms in C++*, Wiley, 2004
The Role of Computing in Engineering

Computing is ubiquitous in engineering. Why?

- Awesome speed of modern, everyday computers makes complicated analysis and simulation possible across all domains.

- Applications in design and modelling. Far beyond the reach of the mortal human engineer. Indeed many modelling problems are utterly infeasible without modern computers and software.

- In embedded systems, computers can provide a level of power, speed, flexibility and control not otherwise possible (e.g., mobile phone)

- Computing is “cheap” (but exercise this argument with care)

- Software is the key

Some examples ...
Example: mobile phone

- Even simple mobile phones rely on software
- Typical phone has a microcontroller (SIM card) with a small program
  - Drive GUI
  - Control devices (keypad, microphone, a/d, dsp, decoder)
Example: Sizewell B

- Nuclear power station (PWR), on-stream in 1995
- Software used extensively in the design
- Software for control!
  - First UK reactor to use software in its Primary Protection System
Example: A380

- 1400 separate programs
- There is a software project just to manage all the software!
- Clearly safety-critical features of the software
Example: NPfIT

- NHS National Plan for IT
- Plan to provide electronic care records for patients
- Connect 30000 GPs and 300 hospitals
- Provide secure access to records for healthcare professionals
- Provide access for patients to their own records via “Healthspace”
Software Engineering vs Programming

- Software engineering is about more than just programming/coding
- It is about design principles and methodologies that yield programs that are
  - Robust
  - Manageable
  - Reusable
Intrinsic difficulties with software

- Analogous vs discrete

- The “curse” of flexibility
  - Can encourage unnecessary complexity
  - Redefinition of tasks late in development – shifting goal-post

- Complexity and invisible interfaces
  - Standard way of dealing with complexity is via modularity
  - But this alone is not enough because interfaces can be subtle and invisible, and here too there is a need to control complexity

- Historical usage information
  - Unlike physical systems, there is a limited amount of experience about standard designs
When software projects go wrong

- A320, Habsheim and Strasbourg
When software projects go wrong

- **London Ambulance Service**
  - 1992, computerised ambulance despatch system fails

- **Therac-25**
  - 2 people died and several others exposed to dangerous levels of radiation because of software flaws in radiotherapy device

- **OSIRIS**
  - £5M University financial package
  - Expenditure to date more like £20-25M

- **NPfIT?**
  - NHS £12 billion IT project

- comp.risks is a great source of others...
NHS National programme for IT: NPfIT

- Plan to **provide electronic care records** for patients
  - Connect 30000 GPs and 300 hospitals
  - Provide secure access to records for healthcare professionals
  - Provide access for patients to their own records via “Healthspace”

- **Laudable?**

- **Realistic?**
  - Software Engineering specialists have their doubts
  - Ross Anderson (Prof of Security Engineering, Cambridge Computing Laboratory) writes in his blog “I fear the whole project will just continue on its slow slide towards becoming the biggest IT disaster ever”.

Software vs “other” engineering

• How is software engineering similar to other engineering?

• **Abstraction** and **Modularity**
  - Consider free-body diagram
  - Thevenin/Norton
  - Low output impedance / High input impedance
  - Digital computer

• We return to these concepts later…
Abstraction: free-body diagram

Block A
- T = 49.05 N
- Fr

Block B
- T = 19.62 N
Modularity: Op-amp buffer

- Unity gain buffer
- $V_{out} = V_{in}$
- Very high input impedance, very low output impedance
Software vs “other” engineering

How is software different to other engineering?
- Pure, weightless, flexible
- Capacity to incorporate massive complexity
- No manufacturing defects, corrosion, ageing
Software life-cycle

- **Software development stages**
  - Specification
  - Design
  - Implementation
  - Integration
  - Validation
  - Operation/Maintenance/Evolution

- Different types of system organise these generic activities in different ways

- Waterfall approach treats them as distinct stages to be signed off chronologically

- In practice usually an iteration of various steps
Requirements

- Vague initial goals
- Iterative refinement
- Leading to more precise specification

Example
- Calculate the n-bounce trajectory of a lossy bouncing ball.
- Refine this to consider
  - What does the statement actually mean?
  - Physics
  - Initial conditions
  - Air-resistance?
  - Stopping criterion (criteria)?
- Now, think about how to design/implement
Validation/Verification

- Verification: does the system confirm to spec?
- Validation: does it actually do what it was supposed to?
- Top-down vs bottom-up testing
- Black-box vs white-box testing
- Impossibility of exhaustive testing
Extreme programming (XP)

- Proposed in the late 90s as a reaction to problems with “traditional” development processes
- Takes extreme position compared with waterfall approach
- Appropriate for small-medium sized projects
  - Teams of pairs of programmer, programming together
  - Incremental development, frequent system releases
  - Code constantly refined, improved, made as simple as possible
  - Do not design for change; instead change reactively
Top-down design

• Here want to keep in mind the general principles
  - Abstraction
  - Modularity

• Architectural design
  - identifying the building blocks

• Abstract specification
  - describe the data/functions and their constraints

• Interfaces
  - define how the modules fit together

• Component design
  - recursively design each block
• Procedural programming: tend to focus on algorithms

• Object-oriented programming: tend to focus on data structures
Structured programming

Top-down vs bottom-up

• Both are useful as a means to understand the relations between high-level and low-level views of a program

• **Top-down**
  - Code high level parts using “stubs” with assumed functionality for low-level dependencies
  - Iteratively descend to lower-level modules

• **Bottom-up**
  - Code and test each low-level component
  - Need “test harness” so that low-level can be tested in its correct context
  - Integrate components

• Not hard-fast rules; combination often best
Simple design tools

Flow chart:

Start

- Alarm Rings
  - Delay: Set for 5 Min.
  - Ready to Get Up?
    - NO: Hit Snooze Button (Average 3 Times)
    - YES: Climb Out of Bed

End

Pseudo-code:

Wait for alarm
Count = 1
While (not ready to get up and count <= 3)
  Hit snooze button
  Increment count
Climb out of bed
Data flows

- Data flow diagram
- Simple example, VTOL simulator

![Data flow diagram]

Controller

Simulator

Display

state

thrust

velocity

Thrust

height

Weight
Simple design tools

State Diagram

- Full power
  - do: set power = 600
- Full power
  - Timer
- Full power
  - Number
- Half power
  - Timer
- Half power
  - do: set power = 300
- Half power
  - Door open
- Operation
  - do: operate oven
  - Cancel
- Enabled
  - do: display 'Ready'
  - Door open
- Disabled
  - do: display 'Waiting'
- Waiting
  - do: display time
In which we consider basic programming constructions …
Basic coding techniques: flow control

- Pretty much any program can be specified using:
  
  - Sequences of instructions
    - `{ Do A; Do B; Do C; }`

- Conditional instructions
  - `If (condition) Do A`

- Repetitions (loops)
  - `While (condition) Do A`

- These *semantic* concepts are implemented in different high-level programming languages using different *syntax*
Basic coding techniques: variables

- Pretty much any program that does something useful requires the use of variables:
  - A symbolic name associated with a data storage location and its contents in memory
  - Value (contents) generally changes during the course of program execution.

- **Assignment**
  - `<Variable> = <Expression> ;`

- **Declaration**
  - `<Type> <Variable> ;`

- Again, the idea of a variable is a semantic concept which is implemented in differently in different high-level programming languages
Addresses

00000000
00000001
...

Memory

0x66
0x72
0x65
0x64

zoom: 8 bytes

Interpretations

integer 0x66726564 = 1718773092
character 66 72 65 64 = “fred”
floating point 1.68302e+022
MATLAB

N = 10;
tot = 0;
totsq = 0;

for i=1:N
    tot = tot+i;
    totsq = totsq + i^2;
end

tot
totsq

C

int N = 10;
int tot = 0;
int totsq = 0;
int i;

for (i=1; i<N; i++) {
    tot += i;
    totsq += i*i;
}

printf("%d\n",tot) ;
printf("%d\n",totsq) ;
Notes on coding style

• Prose
  - Use meaningful **variable names**
  - Use **comments** to supplement the meaning
  - **Indent** code for each block/loop

• Grouping
  - Encapsulate groups of statements sensibly in functions
  - Encapsulate related data sensibly in data structures

• Top-down vs bottom-up
  - **Design** top-down
  - **Code** bottom-up or top-down, or a combination
MATLAB vs C

- MATLAB and C are both **procedural languages**
- MATLAB is an **interpreted** language
  - each statement decoded and executed in turn
- C is a **compiled** language
  - each module (\*.c file) is converted into assembly language
  - The interfaces between the modules are
    - Shared global data
    - Function calls from one module to another
  - This is resolved at link time when the modules are linked together into an executable
Procedural programming

- Aim is to break program down into functional units
  - procedures or functions
  - set of inputs, set of outputs

- In MATLAB and C this procedural building block is the function

- Understanding functions …
Organisation of MATLAB programs

- A MATLAB “program” may be a script or function
  - *i.e.* a sequence of instructions

- This script or function will typically call a bunch of other functions

- Functions are stored in `.m` files
  - Multiple functions can be stored in one `.m` file
  - The first function is visible outside
  - The others are local functions

- Part of the recursive subdivision of the problem
function y = func(x)
y = f(x + 1) ;

function y = f(x)
y = sqrt(x) + f(x) ;

function t = g(omega)
t = sin(omega) ;

function b = foo(a)
b = h(a - 1, 3) ;

function y = f(x, z)
y = bar(x).^z ;

function m = bar(n)
m = n * (n + 1) / 2 ;
Organisation of C programs

- **modulea.h**
  - func
  - #include

- **moduleb.h**
  - foo

- **modulec.h**
  - bar

- **modulea.c**
  - func
  - f()
  - g()

- **moduleb.c**
  - foo
  - f()

- **modulec.c**
  - bar
B16 Software Engineering
Structured Programming
Lecture 3

Functions
Functions

• Function definition
• Function call
• Function prototype
• Scope (local versus global data)
• Parameters and return value(s)
• Low-level implementation of function calls
• Recursion
Function definition

% compute factorial
function z = fact(n)

% function body
z = 1;
for i=1:n
    z = z*i;
end

int fact(int n)
{
    int i, val = 1;
    for (i=1; i<=n; i++) {
        val *= i;
    }
    return val;
}
Function call

• Distinguish between
  - The function definition
    ▪ Defines the set of operations that will be executed when the function is called
    ▪ The inputs
    ▪ The outputs

• And the function call
  - i.e. actually using the function

• Formal vs Actual parameters

• Return value(s)
  - The value of a function evaluation is the return value

```plaintext
fact(10)
a = 6;
z = fact(a);
[V,D] = eig(A);
```
Function prototype

- The function prototype provides enough information to the compiler so that it can check that it is being called correctly
- Defines the interface
- Input (parameter), output (return value)

**myexp.h file**

```c
float myexp(float x);
```

**myexp.c file**

```c
float myexp(float x) {
    const float precision = 1.0e-6;
    float term=1.0, res=0.0;
    int i=0;
    while (fabs(term)>precision) {
        res += term;
        i++;
        term = pow(x,i)/fact(i);
    }
    return res;
}
```
Scope: local variables

- Variables which are declared inside a function are **local variables**

- They **cannot be “seen” outside the function** (block) in which they are declared

- A local variable **exists only for the duration of the current function execution**
  - It is declared as a new variable every time the function is called
  - It ceases to exist when the function returns
  - It does not “remember” its value between calls
**Scope: global variables**

- **Global variables** exist outside all functions
- A global variable is **visible inside all functions**
- If there exist two variables, one local, one global, with the same name, then **the local one takes precedence** within its local scope
- C and MATLAB behave differently
  - C will use a global if no local exists
  - MATLAB only uses a global if the programmer explicitly requests it
- **Globals should be used with caution because their use inside a function compromises its encapsulation**
Encapsulation

• Want the function to behave in the same way for the same inputs
  - encapsulate particular functional relationship

• But if the function depends on a global it could behave differently for the same inputs

• Live example using myexp
Function encapsulation

input parameters → function → output values

hidden input → global var. → hidden output
Side-effects

- Could **set value of a global variable in a function**

  - This compromises the function’s encapsulation
    - causes a side-effect
    - an implicit output, not captured by the interface
    - makes it difficult to re-use code with confidence

- Consider for example C and MATLAB **function libraries**
  - set of re-usable routines with well defined interfaces

- In small projects maybe not a big problem

- Hugely problematic in bigger projects, especially when multiple programmers working as a team
  - complicates interfaces between components, possibly in unintended ways
Low-level implementation of function call

Addresses
00000000
00000001
...

Memory

CODE

DATA

STACK

machine code

global variables

Function Activation Stack Frame

local variable $m$

...

local variable 1

return location

parameter $k$

...

parameter 1

return value $n$

return value 1
Libraries

• Well developed, generally useful functions are typically placed in libraries.

• **Example** `libm.a`

  `libm.a` is the C math library
  
  - contains well-tested, efficient implementations of mathematical functions like `sin()`, `cos()`, `sqrt()`, etc.

• The function prototypes are in `math.h`
  
  - this is all the programmer needs to know
  - the function implementation detail is hidden from the programmer.

• **Example** MATLAB toolboxes
Pass by value

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
}

Output
5
10

Pass by reference (C++)

void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
}

Output
10
5
Pass by value

void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
}

Pass by reference (C++)

void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
    return;
}

Caller
int i = 5, j = 10;
swap(i, j);
cout<<i<<endl;
cout<<j<<endl;

stack

stack
int i = 5, j = 10;
swap(&i,&j);
printf("%d %d\n", i, j);

void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
    return;
}
Passing functions as parameters

- Consider an algorithm for numerical solution of a differential equation
  \[ \dot{Y}(t) = -Y(t) \]

- The code may look like this

  ```matlab
  function Y = Euler(Y0, h, N)
  Y = zeros(numel(Y0), N);
  for n = 2:N
    Ydot = - Y(:, n-1);
    Y(:, n) = Y(:, n-1) + h * dotY;
  end
  ```

- But we don’t want to have to re-write the code for every function we want to find the roots of!
Passing functions as parameters

More generally:
\[ \dot{Y} = f(Y, t) \]

What is f and can we encapsulate the Euler method without building f into it?

function \( Y = \text{Euler}(f, Y0, h, N) \)

\[
Y = \text{zeros}(	ext{numel}(Y0), N);
\]

\[
\text{for } n = 2:N
\]

\[
  t = h * n ;
\]

\[
  Ydot = f(Y(:,n-1), t) ;
\]

\[
  Y(:,n) = Y(:,n-1) + h * \text{dotY} ;
\]

end

f is a handle to a user-defined function
- that takes current value of \( Y \)
- returns \( f(Y, t) \)

```
Example usage
Euler(@myODE, Y0, h, N) ;

function myODE(Y,t)
  dotY = -Y ;
end
```

With anonymous functions

\[
f = @(Y,t) - Y ;
\]

Euler(f, Y0, h, N) ;
Passing functions as parameters

- C code to contrast (simplified version):

```c
float myODE(float Y, float t)
{
    return - Y ;
}

float Euler(float func(float, float), float Y0, float h)
{
    return Y0 + h * func(Y0, t);
}

int main()
{
    int n ;
    float Y[200], h = 0.05;
    Y[0] = 1.0;
    for (n = 1 ; n < 200 ; n++) {
        Y[i] = Euler(myODE, Y[n-1], h);
        printf("Y[\%d] = \%f\n", t, Y[n]);
    }
}
```
Recursion

• Recursion is the programming analogue of induction:
  - if \( p(0) \) and for all \( n \geq 0 : p(n) \) implies \( p(n+1) \)
  - then \( p(n) \) for all \( n \geq 0 \)

• Define a function in terms of:
  - itself
  - boundary conditions

• **Example**: factorial (\(!\))
  - \( n! = n \times (n - 1)! \)
  - \( 0! = 1 \)
Live demo

```matlab
function y = factorial(n)
if n==0 || n==1
    y = 1;
else
    y = n * factorial(n-1);
end
```
Recursion: example 2

• Multiple recursion

```c
const int SIZE = 256;
int im[SIZE][SIZE];

void fill (int x, int y, int old_colour, int new_colour)
{
    if (x>=0 && x<SIZE && y>=0 && y<SIZE) {
        if (im[y][x] == old_colour) {
            im[y][x] = new_colour;
            fill (x-1,y,old_colour,new_colour);
            fill (x+1,y,old_colour,new_colour);
            fill (x,y-1,old_colour,new_colour);
            fill (x,y+1,old_colour,new_colour);
        }
    }
    return;
}
```
Data types and data structures

- C/C++ predefine a set of **atomic types**
  - `bool`, `char`, `int`, `float`, `double`

- C/C++ provides mechanism for building **compound data types**
  - `struct`
  - `class`
  - `array`

- MATLAB supports
  - arrays/matrices (of course)
  - structures
  - classes
C/C++ struct (and class)

- A struct (class in C++) is a compound data type which encapsulates related data into a single entity

**Declaration**

```c
% C-style declaration
struct Complex_ {
    double re ;
    double im ;
} ;

% Shorter definitions: typedef
typedef Complex_ Complex ;
typedef struct {
    double re ;
    double im ;
} Complex ;
```

**Definition**

```c
struct Complex_ c ;
c.re = 1.0 ;
c.im = 0.0 ;

% With typedef
Complex c ;
```
Example: VTOL state

- Represent current state as, say, a triple of numbers and a bool, (position, velocity, mass, landed)

- A single struct variable represents all numbers
  - data encapsulation
  - better abstraction

- **Declaration**
  ```
  typedef struct {
      double pos;
      double vel;
      double mass;
      bool landed;
  } State;
  ```

- **Definition**
  ```
  State s;
  ```
Accessing struct/class members

typedef struct {
  double pos ;
  double vel ;
  double mass;
  bool landed;
} State ;

State s;

s.pos = 1.0;
s.vel = -20.0;
s.mass = 1000.0;
s.landed = false;

thrust = ComputeThrust(s) ;
typedef struct {
  double pos;
  double vel;
  double mass;
  bool landed;
} State;

State s;

s.pos = 1.0;
{s.vel = -20.0;
  s.mass = 1000.0;
  s.landed = false;
}

thrust = ComputeThrust(s);

The closest thing that MATLAB has to a declaration is a class definition

classdef State
  properties
    pos;
    vel;
  end
end

s = State();
{s.pos = 1.0;
  s.vel = -20.0;
  ...
  Thrust = ComputeThrust(s);
Functions
Output parameters

Image ReadImage (const string filename, bool& flag);
bool ReadImage (const string filename, Image& im);

• Input
  - filename (type string)

• Output
  - im: an image (value of type Image)
  - flag: a boolean indicating success/failure

function [Image, flag] = ReadImage(filename)

• Basically the same, but cleaner in MATLAB!
Arrays

An array is a data structure containing a numbered (indexed) collection of items of a single data type.

% An array of integer variables
int a[10];
res = a[0] + a[1] + a[2];

% An array of Complex variables
Complex z[20];

% An array of State variables
State s[100];

for (t = 1 ; t < 100 ; t++) {
    s[t].pos = s[t-1].pos + s[t-1].vel + 0.5*g;
    s[t].vel = s[t-1].vel + g - GetThrust(s[t-1], burnRate)/s[t-1].mass;
    s[t].mass = s[t-1].mass - burnRate * escapeVel;
}
Multi-dimensional arrays

double d[4][5] ;

has elements:

\[
\begin{bmatrix}
  d[0][0] & d[0][1] & d[0][2] & d[0][3] & d[0][4] \\
\end{bmatrix}
\]
Dynamic structures

• C/C++ arrays are **static structures** that live either in global memory space or on the stack.

• It is often useful to be able to **create space for variables (objects) dynamically at runtime**

• This is done in a memory area known as the **heap**

• Useful dynamic structures
  - Linked lists
  - Trees
  - Graphs
Pointers and dynamic memory allocation

- pointer = variable holding an address
- The **heap** is an area of memory that can be allocated/freed at run-time by the program.

```c
% get four bytes on the heap
int* aPointer = malloc(4);  
*aPointer = 77;
```
Dynamic memory in C

- In C dynamic (runtime) memory management is performed using:
  - `malloc()`
  - `free()`

- Allocating a block of memory requires knowing its size in bytes:
  - Each data type has a different size:
    - *e.g.* usually `char` = 1 byte, `short` = 2 bytes, `int` = 4 bytes, etc.
  - How to make sure?
  - What about compound types?

- Use the `sizeof()` operator:

  ```c
  DataStructure *d;
  DataStructure darray[];

  d = malloc(sizeof(DataStructure)); % allocate 1 DataStructure val.
  Darray = calloc(N, sizeof(DataStructure)); % allocate N
  ```
% a list element type

typedef struct ListElement_ {
    struct ListElement_ *next;
    int value ;
} ListElement ;

% add a new element after current
ListElement *newElement = (ListElement*)malloc(sizeof(ListElement));

newElement->next = current->next ;
current->next = newElement ;

% delete the element after current
ListElement *temp = current->next;
current->next = temp->next;
free(temp);
• Consider binary tree

typedef struct {
    Data d;
    struct node *left;
    struct node *right;
} Node ;

• Depth first traversal
Graphs

- A graph comprises a set of nodes and connecting edges

```matlab
classdef graph
    properties
        nodes
        edges
        adjacency
    end
end
```
Algorithms

- A precise set of instructions (a “recipe”) for how to solve some problem

- We’ll look at some **basic algorithms**
  - sorting
  - searching
  - numerical computing

- We’ll also look at:
  - The complexity **order of an algorithm**
  - Aspects of formal proofs of algorithms, especially the idea known as loop invariants
A common task is to sort a set of objects 

Suppose we have a set of $N$ objects where each object $i = 0, ..., N - 1$ has some property or key $a[i]$ associated with it which can be assessed as larger or smaller than some other object’s key $a[j]$.

Then sorting into ascending order consists of relabelling the objects such that the keys are sorted:

$$a[0] \leq a[1] \leq a[2] \leq \ldots \leq a[N - 1]$$
Insertion sort

- **Algorithm**
  - Remove an element from the unsorted list (*e.g.* first card in a pack)
  - Insert the element in the sorted list at the correct place

- Simple to code and quite efficient for small arrays/lists
Insertion sort

- Pseudo code:

```
%input: a[1], ..., a[N]
for i = 2 to N
    key = A[i]; % remove element key
    % shift the sublist a[1 ... i-1] by one until
    % key can be put back
    j = i - 1;
    while (j >= 1 && key < A[j])
        j = j - 1;
    endwhile
    % put key back
    A[j] = key;
endfor
```
Order of an algorithm

- Characterize functions according to their growth rates
  - different functions with the same growth rate represented using the same order

- "Big O" notation
  - describes the limiting behaviour of a function when the size of the arguments
tend towards a particular value (usually infinity)

- **Insertion sort**
  - Loop \( N \) times
  - Within each loop we do 1, then 2, then 3, etc. operations
  - Total operations
    \[
    \sum_{n=1}^{N} n = \frac{N(N - 1)}{2}
    \]
  - \( O(N^2) \)
Divide and conquer

- Divide and conquer is a recursive paradigm:

  - Solve(\textit{problem})
    - If problem difficult:
      - Subdivide \textit{problem} into \textit{subproblems}
      - For each \textit{subproblem} Solve(\textit{subproblem})
      - Combine \textit{subproblem} solutions
    - Return solution

- Note the \textit{recursive call}

- Some of the best known and most famous (and useful) algorithms are of this form, notably quicksort and FFT
Complexity analysis of divide and conquer

- Assume linear work is done for each subproblem

- Number and size of subproblems to solve
  - $N$ items in first subproblem = $O(N)$
  - $N/2$ items in 2 subproblems = $O(N)$
  - $N/4$ items in 4 subproblems = $O(N)$
  - ...
  - 1 item in $N$ subproblems = $O(N)$

- depth = $\log_2(N)$

- total: $O(N \log_2(N))$
Merge sort
Merge sort

**MERGE-SORT**(list)

if length(list) > 2 then
  Split list into *firsthalf* and *remainder*
  MERGE-SORT(*firsthalf*)
  MERGE-SORT(*remainder*)
  % combine *firsthalf* and *remainder* to give ordered list
  list = MERGE-SORTEDLISTS(*firsthalf*, *remainder*)
else if length(list) == 2 then
  % Swap if needed ...
  if (listhead > listtail) then
    temp = listhead
    listhead = listtail
    listtail = temp
  end if
else
  % Do nothing at all. The list with one element is sorted!
end if
return (list)
Insertion versus Merge Sort

InsertSort vs MergeSort

Time to Sort (sec)

Size of List
Root finding

- Find zeros of a non-linear (scalar) function of one variable

- **bisect**($f$, $x_1$, $x_2$)
  
  $μ = (x_1 + x_2) / 2$
  
  if $f(μ)$ close to zero then return $μ$

  else
  
  if $f(μ) * f(x_1) > 0$
    
    return **bisect**($f$, $μ$, $x_2$)

  else
    
    return **bisect**($f$, $x_1$, $μ$)
Program correctness: Loop invariants

- The key to understanding many algorithms is understanding what action a loop or loops are performing

- A loop invariant is
  - a relation among program variables
  - that is true when control enters, executes, and leaves a loop

- Loop invariants help analyze programs, check for errors, and derive programs from specifications.

- The desired outcome of an algorithm can often be specified by a loop invariant and a terminating criterion (post-condition)

- Together with precondition this can specify the entire algorithm
Loop invariants

- General formulation
  - ...

- % the Loop Invariant must be true at the start
  while ( TEST CONDITION ) {
    % Loop Invariant must be true here
    ...
  }
  % Termination + Loop Invariant = Goal

Example: Consider insertion sort
• **Example** insertion sort

• for i=2 to N
  % Invariant 1:
  % a[1...i-1] is a sorted permutation of the original a[1...i-1]
  key = A[i];
  j = i-1;
  while (j>=1 && key<A[j])
    % Invariant 2: A[j ... i] are each >= key
    j=j-1;
  end
  A[j] = key;
end
% Termination: i = N + 1
Consider we have **binary search tree**, *i.e.*:

- the **left subtree** of a node contains only nodes with keys less than the node's key
- the **right subtree** of a node contains only nodes with keys greater than the node's key
- both left and right subtrees are also binary search trees.

Node * **BinarySearchTreeSearch**(Node * node, double key)
{
    if (n == NULL)  
        return NULL ;
    else if (k < n->d.key)  // n->d is same as (*n).d
        return TreeSearch(n->left, k);
    else if (k > n->d.key)
        return TreeSearch(n->right, k);
    else
        return n;
}

Complexity is $O(\log N)$
Minimum spanning tree

See tute sheet
Concept summary

- **Software engineering principles**
  - Abstraction
  - Encapsulation
  - Modularity

- **Managing complexity**

- **Functions**
  - Encapsulation of related instructions to create “Black-box” functionality
  - Code re-use

- **Data structures**
  - Encapsulation of semantically related data together into appropriate structures

- **Algorithms**
  - Designing algorithms
  - Standard algorithms
  - Loop invariants

- Exam questions? See tutorial sheet to follow.