How do we know that an algorithm solves a particular problem?

1 Software Bugs

- **Software Bug** - An error in the program that causes the program to behave in an unintended or an unexpected way
- This usually arises from the programmer's error, but can arise because of a "secondary error" - such as a poorly written complier

Bugs can affect programs in various ways

- Produce incorrect results
- Program crashes
- Susceptible to malicious attack
- Runs slowly or gobbles up memory

2 How do we get bug free software?

- **Software Engineering** - The application of a systematic, disciplined and quantifiable approach to the development, operation and maintenance of the software

The 5 phases of the waterfall model:

- **requirements phase** - where the specification of what the program is intended to do is produced
- **design phase** - where a plan is composed for the intended solution and where this plan involves algorithms and their implementations, architectural aspects, data structures and so on
- **implementation phase** - where the design is implemented as code
- **verification phase** - where the code is tested and debugged
- **maintenance phase** - the continual updating and modifying of the code

3 Establishing correctness: software testing vs formal methods

Software testing - The process of evaluating an attribute or capability of a program or system
The phases of the process of software testing:

- Modelling the program's environment so that the program will be tested under conditions as near as possible to those for which the program has been designed
- Selecting test scenarios so as to try and achieve as good a coverage as possible
- Running and evaluating test scenarios, preferably in an automated fashion
- Measuring test progress

The problem with this method is that it depends on the tests you pick, there may be a set of inputs that you haven't used that would cause an error.

One way to test this is through an alpha or beta test, this will mean that users will do huge sets of inputs, and will likely find any bugs that exist

Formal Methods: Use mathematically based techniques to formally specify and formally verify software and hardware systems

- More concerned with correctness than implementation
- Two fundamental activities - specification and verification

- The specification is derived as a collection of logical formulae
- Mathematical techniques are sometimes used to convert the specification into a program so that it can be verified mathematically
- Formal methods are closely linked with automated theorem proving, where programs are developed to automatically prove mathematical theorems
- They are also linked to model checking, where properties of specifications are automatically evaluated using model checking software

Some argue that the expense, time and skills required for formal techniques are not merited. However if safety critical operations, formal methods will give the more accurate result, so is the better option.

Software metric - A measure if testing hardness

- How sensible/efficient do you want code?
- How much time and money to spend before giving up?
- Two main metrics
  - Lines of code - only eliminate a certain number of bugs per certain lines of code
  - Cyclotomic complexity - How many paths through the code are there? Find core paths and get them bug free, more unused paths may have some bugs left in

4 What do we mean by correct?

Totally Correct - Correct with respect to some specification and terminates on every input
Partially Correct - Only correct with respect to the specification, no guarantee of termination

Assertions are used to check if conditions hold of variables etc, for example if a variable 0 when it should be etc.

5 Factorial Induction

5.1 Recursion

- **Base case** - When the value of y is 0 - terminates and is correct
- **Inductive step**
  - Assume P(n) for some $n \geqslant 0$, that is, factorial(y) terminates and is correct whenever we have the input value os y is $\geqslant 0$ and $\leqslant$ n
  - Factorial(y) with the input value of y given as $(n + 1) > 0$
    - Induction Holds yeilds termination and that it computes $(n + 1)!$

5.2 Iteration

- **Termination**
  - Observation: y decreases by 1 at the end of every iteration of the while loop
  - So there are n iterations of the while loop and we have termination
- **Correctness**
  - n-y+1 increases by 1 at the end of each iteration of the while loop
  - Prior to the first execution of the while loop n-y+1 has value 1
  - observation: during the ith iteration, x is multiplied by y
  - Our algorithm outputs n! after n iterations

6 More complicated inductions

Consider $\Pi = \{(a,b) : a \geqslant b \geqslant 0\}$
Define the lexicographic ordering on this set Π

- $(a,b) = (c,d)$ if, and only if, a=c and b=d
- $(a,b) > (c,d)$ if, and only if, either $a > c$ or ($a = candb > d$)

So, the pairs in Π can be ordered as

$$(0,0) < (1,0) < (1,1) < (2,0) < (2,1) < (2,2) < (3,0) \\ < (3,1) < (3,2) < (3,3) < (4,0) < \dots$$

Suppose that some property P is indexed by pairs $(a,b) \in \Pi$. We can undertake induction as follows

- **Base Case**: Show that $P(0,0)$ holds (or in general, $P(a_0,b_0)$, for some $a_0 \geqslant b_0 \geqslant 0$)
- **Inductive step:** Assume that $P(a,b)$ is true for some (a,b) where $a \geqslant b \geqslant 0$ (in general, where $(a,b) \geqslant (a_0,b_0)$ in the lexiographic ordering). We then prove that if this is the case then $P(a_+,b_+)$ is the next pair in the lexiographic ordering after (a,b)
- **Principle of induction:** allows us to state that $P(a,b)$ holds for all $a \geqslant b \geqslant 0$ (in general, when $(a,b) \geqslant (a_0,b_0)$)

There is nothing clever going on here. What we are really doing is working with the property $Q(n)$ which is the property '$P(a,b)$ where (a,b) is the nth pair in the lexiographic ordering of Π'

6.1 In lecture

-

7 Euclid's algorithm

7.1 Recursively

Recall the recursive version of Euclid's algorithm, namely Euclid(x,y), where we assume that initially x=m and y=n with $(m,n) \in \Pi \backslash \{(0,0)\}$

```
algorithm:  Euclid(x, y)
IF y == 0:
  return x
ELSE:
  set x' = y and y' = x mod y
  set x = x' and y = y'
  Euclid(x, y)
```

We can use our extended principle of induction to prove total correctness. We would also like to use induction as follows: 'our induction hypothesis tells us that any recursive call to Euclid is totally correct; so, our main call to euclid must be totally correct'
We begin with an observation: if Euclid(x,y) is called with the value of (x,y) being $(a,b) \in \Pi \backslash \{(0,0)\}$ and if Euclid(x,y) is then called recursively with the value of (x,y) being (a',b') (so, necessarily $b > 0$ in order for this recursive call to occur) then we must have that:

7.2 In lecture

- Call euclid with 2 inputs m and n greater than 0 where $a \geqslant b$ and if euclid(m,n) is subsequently called recursively with the value of (m,n) being (a',b')
- $(a',b') > (0,0)$ and $a' \geqslant b'$
- $(a,b) > (a',b')$
- So, the values obtained tracking consecutive values of (m,n) are decreasing and so will eventually reach 0, after which the algorithm will terminate

Induction

- **Base case** - m=1 and n=0 - algorithm terminates and is correct
- **Inductive step: assume P(m,n)**
  - Let (m_+, n_+) be the successor of (m,n) in the lexiographic order
  - If $n_+ = 0$ then we are done
  - If $n_= > 0$ then we make the recursive call of euclid$(n_+, m_+ mod n_+)$ and as these values are no bigger than before, moving towards the base case, and so will terminate correctly

7.3 Iteratively

- **Termination**: Set a check point immediately prior to the execution of the while test
  - At consecutive check points
    - Find the pattern in what happens $(m, n) > (0, 0)$ and $m > n$
    - The value is strictly decreasing
- **Correctness**
  - Suppose that initially m and n are u and v respectively, and m has value w on termination
  - Two possibilities, swap or not, one value reduces and one swaps
  - So an integer x divides both u_1 and v_1 iff x divides both u_2 and v_2
  - Eventually x will divide w and 0, and so the answer will be correct

8 Invariants

- A property that is always true at some given check point in an algorithms execution (usually prior to a loop test)

Easiest to see in rewriting of iterative algorithms
Copy algorithms in