Computational Thinking

**What is Computer Hardware**

A CPU is an integrated circuit (IC) which contains billions of transistors.

Copies of the IC are etched on a piece of silicon called a die. These are tested and the error free dies are cut and mounted in a package with the die’s pads connected to the package’s pins

A multi core processor has multiple independent cores on the same IC

Moore’s Law – Transistor capacity doubles every 18-24 months. This will likely not remain true for much longer. Power dissipation also a problem – power proportional to number of transistors and clock speed.

We use transistors to build logic gates which take 0 or 1 as input and produce 0 or 1 as output.

NOT Gate: Input X, output Z

Z = 1 only if X = 0

AND Gate: Input X, Y, output Z

Z = 1 only if X = 1 and Y = 1

OR Gate: input X, Y, output Z

Z = 1 if X = 1 or Y = 1 or both

NAND Gates can be used to build NOT, AND and OR gates.

Any function f(X1, X2, …, Xn) : {0,1}n -> {0,1} can be computed by a circuit made only from AND, OR and NOT gates.

e.g full/half adders

Parts of CPU

-Datapath – performs data processing operations. Includes ALU and Registers

-Control – tells datapath, memory and IO what to do

-Cache – small, fast memory, but expensive

Von Neumann architecture

-Stored program architecture – memory stores both data and programs

-allows for self-modifying programs

-The Von Neumann bottleneck is the limitation of data transfer between CPU and memory. Adding more cache means more frequently used data can be retrieved without having to transfer from memory, reducing this bottleneck.

Harvard architecture

-Data memory and instruction memory are separated, each with their own buses.

-Modern computers use modified Harvard architecture with a single main memory, but CPU cache separated.

Memory

-“pigeon holes” containing data, each with its own address and hold 8 bits. Larger “words” can be formed by combining bytes.

Cost and performance of memory proportional to its distance from CPU

Buses

-Address bus determines the location (memory address) in memory. The width determines the maximum number of addressable memory locations.

-Data bus carries contents of memory. Width determines the word size – maximum data transferred at the same time.

-Control bus deals with information transfer, e.g. whether data is being read or written and control signals to I/O devices.

Cache

-Expensive memory to store rapidly accessed items

- Level 1 cache is often built into the CPU, small, fast

-Level 2 cache is often in the CPU or a short distance away

Registers are on chip memory locations (faster than cache, used for currently running executions)

Accumulator – register where arithmetic is done

Program counter – Holds the address of the next instruction

Fetch decode fetch execute

Fetch- supply instruction address from program counter to address bus. Memory puts contents of this address onto the data bus. Instruction stored in instruction registers.

Decode – Instruction word stored in registers decoded by CPU to tell ALU what to do. Program counter value pushed onto AB and ALU increments this value by word size and sends it back to the program counter.

Operand fetch – Instruction registers provide addresses of data that needs to be processed to the address bus. Memory supplies this data to the data bus so it can be processed by the CPU

Execute – Processing performed on the operand by the ALU, result stored in accumulator. Program counter updated if a branch instruction is executed.

Write back- The result from the accumulator is written back into memory if necessary.

Multi-Core Computing

-More than one processor usually integrated on one circuit.

-Architecture can be shared memory or distributed memory (where each CPU has its own memory) with an interconnection network.

-Allows for use of more and slower processors, resulting in lower heat and power consumption than a single high-power processor.

How does hardware execute software

The Instruction Set Architecture (ISA) is the interface between hardware and software. The ISA includes everything programmers need to know to write programs for the processor. It includes organisation and structure of programmable storage, the instruction sets and formats, modes of addressing and the process of execution.

A primary component of the ISA is its Assembly language.

-High level programs are compiled into assembly language

-Assembly language instructions are very low level and are assembled into machine code.

I-type instructions have a 6 bit opcode, 5 bits for the source register, 5 bits for the destination register and 16 bit address

R-type instructions have a 6 bit opcode, 5 bits each for two source registers, a destination register and a shift amount, and a 6 bit function (or a supplement for opcode)

J-type instructions have a 6 bit opcode and a 26 bit address.

Operating systems

Operating systems manage all of a computer’s resources. They provide the interface between applications and hardware, and is needed as there are many concurrently executing tasks which need managing.

They abstract the hardware for applications – all applications’ access to hardware is through the OS using ISA instructions and system calls. OS provides data security.

The OS kernel is the main component of the operating system, loaded at boot time.

OS main functions

-virtualisation (disk access, virtual machine)

-**starts and stops programs, frees memory, suspends execution**

**-manages memory (users don’t need to know exact memory locations)**

**-handles I/0 and interrupts**

-Maintains file system and access rights

-Networking and security

-Error handling and recovery

Input/output

-I/O devices such as hard disks, CD, GPU, keyboard etc are connected to CPU with a bus.

-Buses are cheap and versatile, but slow in comparison with the CPUI resulting in bottleneck.

-OS helps alleviate bottleneck and the CPU continues with other tasks. When IO device finishes, it raises an interrupt, handled by interrupt handler in the OS.

Processes

-Process = a program in execution, not a program on the disk. Programs can have multiple processes

-Process = thread(s) + address space

-Multiple threads or processes need to communicate and synchronise

-Mutual exclusion – where 2 threads have access to the same counter

Process life cycle

-Typical abstraction of process states:

- new: been created

- ready: not on CPU but ready to run

- running: executing on CPU

- blocked: waiting for an event and so not runnable

- exit: process finished

-State transitions

- admit – process control setup new -> ready

- dispatch – scheduler gives CPU to runnable process ready -> running

- timeout/yield – running process forced or volunteers to give up CPU running -> ready

- event-wait – process waiting for, e.g I/O, gives up CPU running -> blocked

- event – event occurs, wake up process blocked -> ready

- release – process terminates, release resources running -> exit

(see lecture ppt for FSM of this)

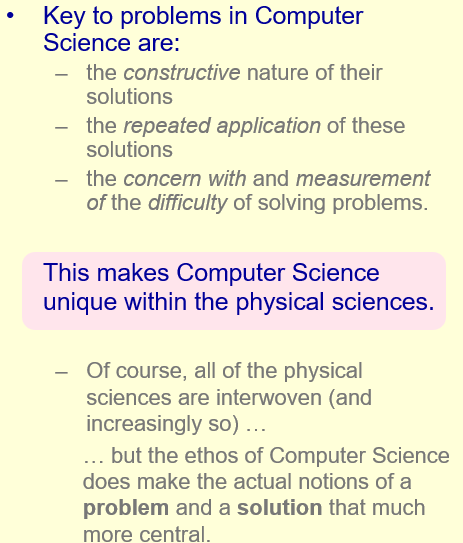
Process control block (PCB) is used by kernel to manage a process. It has a unique process ID, process state, CPU scheduling information and its own program counter and stored values from CPU registers (when not running so values can be restored in CPU when running again), as well as memory management and scheduling/accounting info.

Context switching is how the OS manages multiple processes by constantly saving and restoring data to and from the PCB of each process. This is time consuming.

**Fundamental Problems**

Computer science is the study of computers and what they can do, i.e. the powers, limitations and uses of computers, and their applications to solving problems.

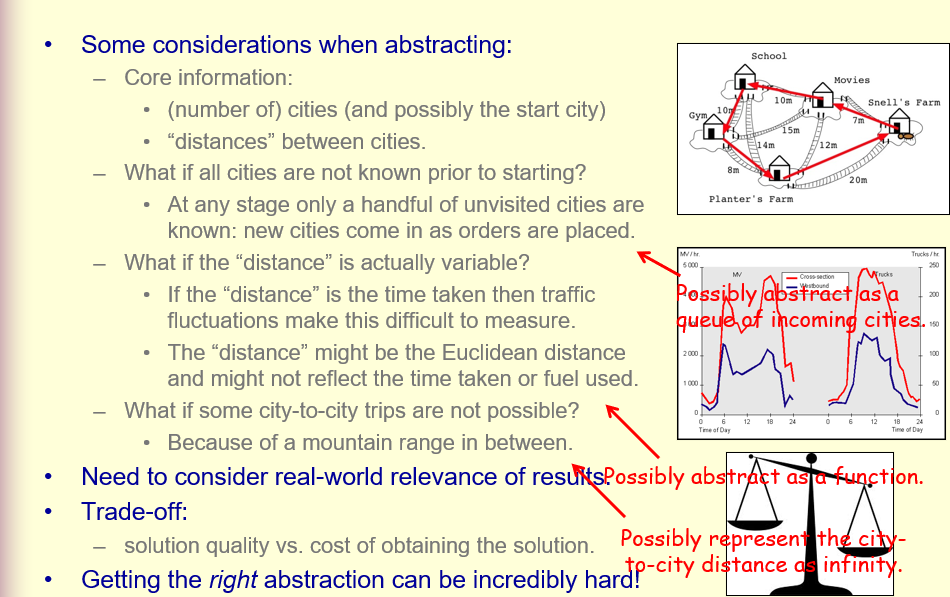
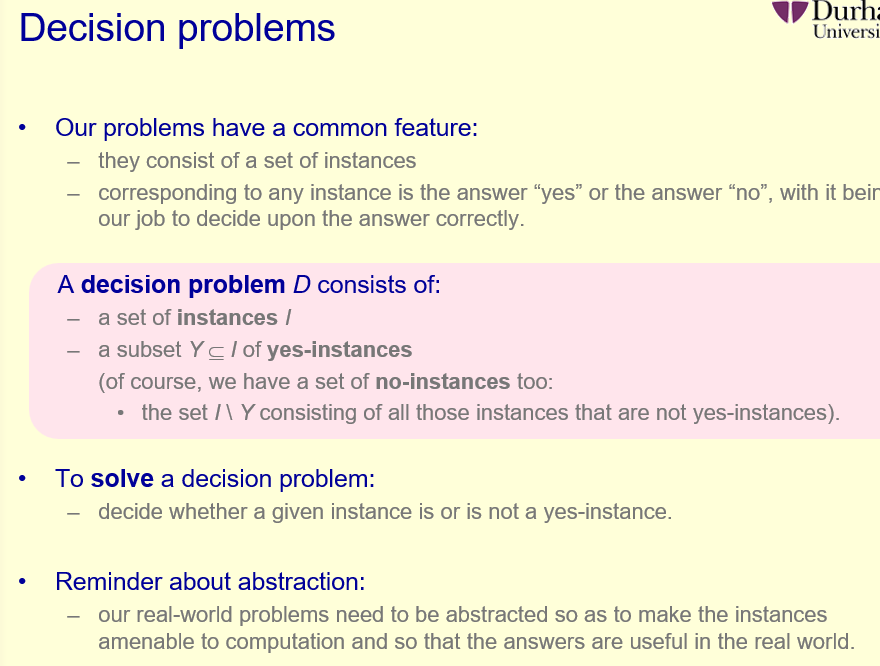
Problems include writing programs without bugs, how to store and lookup data and how to make computers easily programmable.

Problems in CS are primarily about the repeated *journey* towards some goal, rather than just achieving a result in isolation, i.e. more about the method or algorithm than the solution.

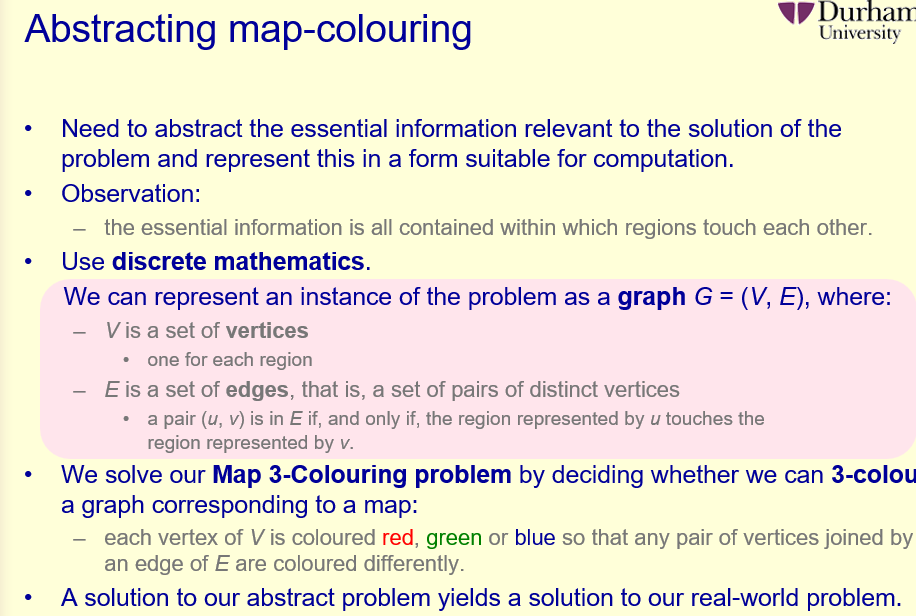
Solutions to CS problems involve some computational device, and demands of the solution have to be met in practice. The notions of computation, resource (i.e. time, memory usage) and correctness are all important. CS is often concerned to finding solutions to real world problems by solving an abstraction of this problem, rather than solving the real world problem itself.

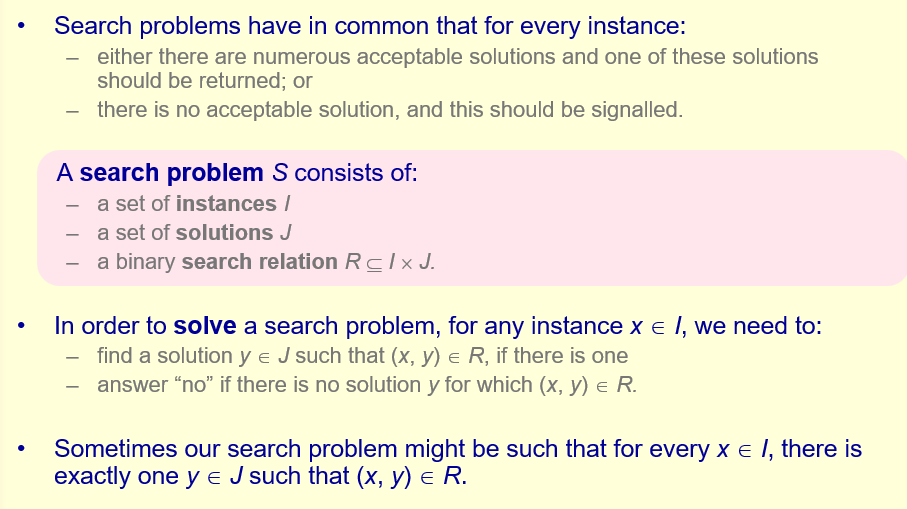
Abstractions of real world problems need to be able to represented and manipulated by a computer while mirroring reality close enough for any solution reached by the computer to remain valid in reality. Abstraction is about finding the level of detail needed to solve the given problem.

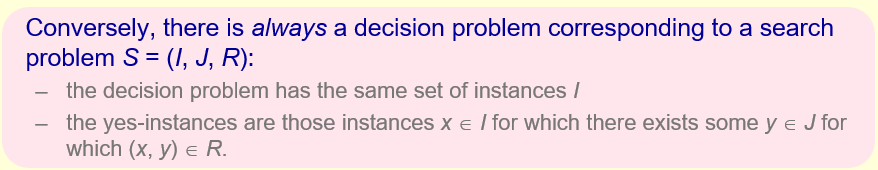
Possible abstractions for travelling salesman problem:

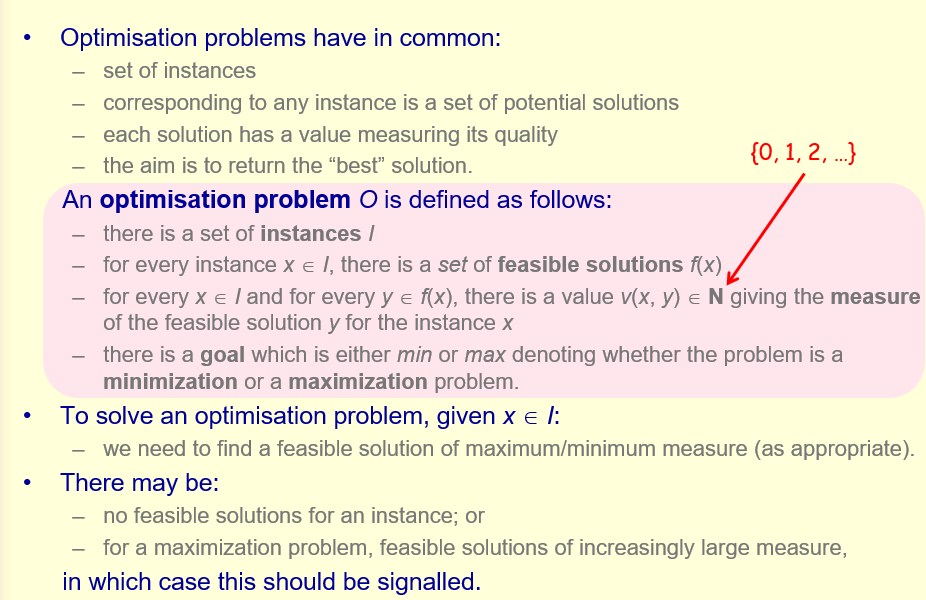
Decision problems:

3-colour map problem (example of a decision problem):

Search problems:

A decision problem usually decides whether or not there is a solution to a given problem – the equivalent search problem returns the solution, rather than just verify that there is one. There is always a decision problem corresponding to a search problem:

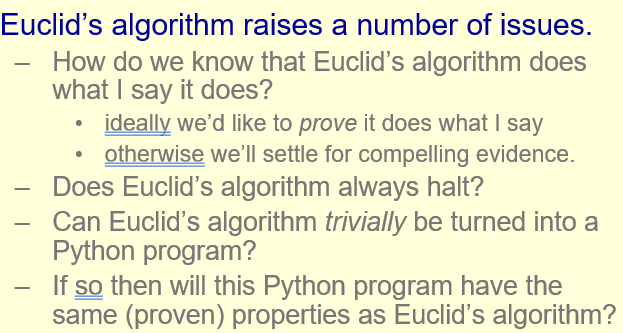
Optimisation problems:

i.e. every possible solution in the set of solutions f(x) is given a value v, called the measure, which represents how “good” of a solution it is. This measure should either be minimal or maximum, depending on the problem, in order to get the most optimal solution, or indicate that there are no possible solutions if applicable.

See lecture slides for example of optimising map colouring 😊

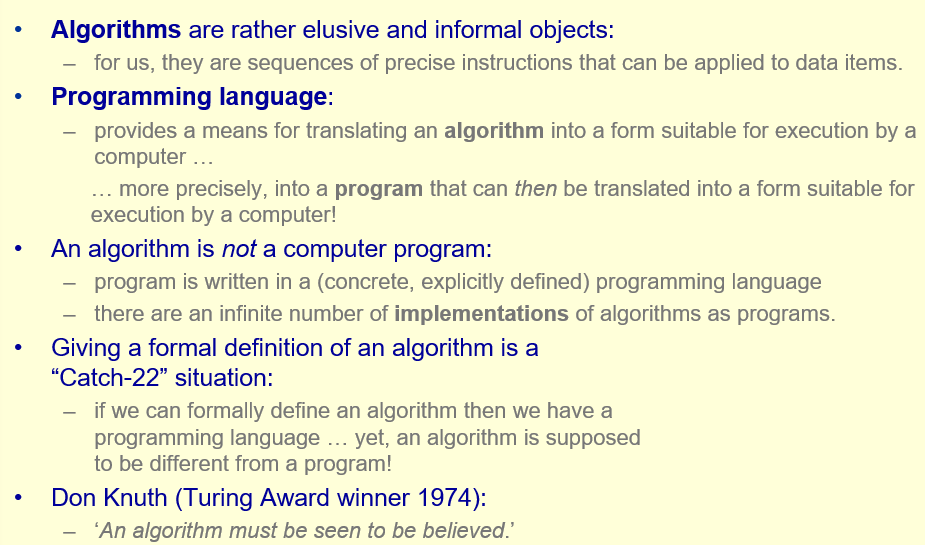
So far, algorithms have been described using natural language and have been run by hand. Algorithms need to be implemented as programs for execution by computers. We need to be able to use more precise language to describe our algorithms, which needs to be precise enough to both analyse algorithm performance and easily implement it in Python, as well as being flexible enough to be able to write implementations in other high level languages. – Pseudocode.

Problems with pseudocode for Euclid’s algorithm (from lecture slides)

Proving a program does what it is meant to in all cases, and proving exactly how long an algorithm takes to do what it does is difficult.

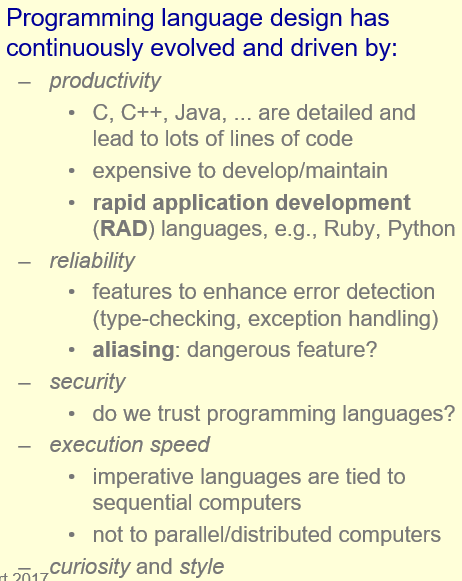
Small pieces of code can be mathematically proved, whereas larger code has to be tested systematically to establish program capability. Testing every possible set of inputs difficult.

Algorithms and Programming Languages



Programming langauges are grouped by paradigms.

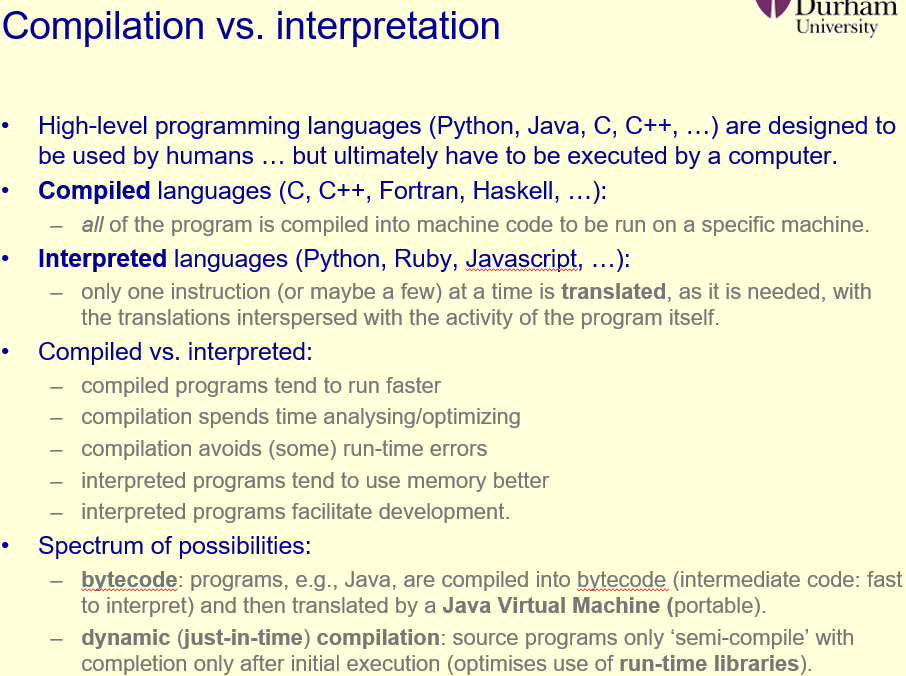
* Imperative – e.g. Python, C++ - statements to change a program’s state.
* Declarative –– saying what to do rather than how to do it. This includes Functional languages, where programs are defined as mathematical functions, and logic languages such as Prolog.
* Data-oriented – e.g. SQL
* Scripting – e.g. Javascript, Perl – designed for automating tasks that involve passing commands to external programs. (Python also does this apparently)
* Assembly – low level languages providing the interface between high level code and machine code.
* Concurrent – e.g. Ada, Erlang – provides facilities for concurrency or shared memory
* Dataflow – e.g. Vizsim, Openwire
* 4th-generation – e.g. SAS, often used commercially and built around databases
* List-based, e.g. Lisp, built around list data structures
* Visual (ew) – ugy images are code apparently

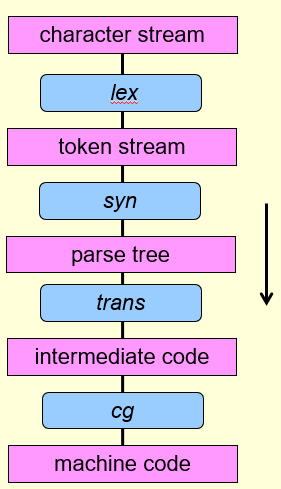
Why so many languages?

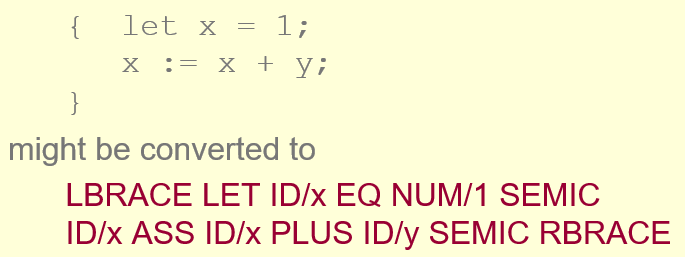
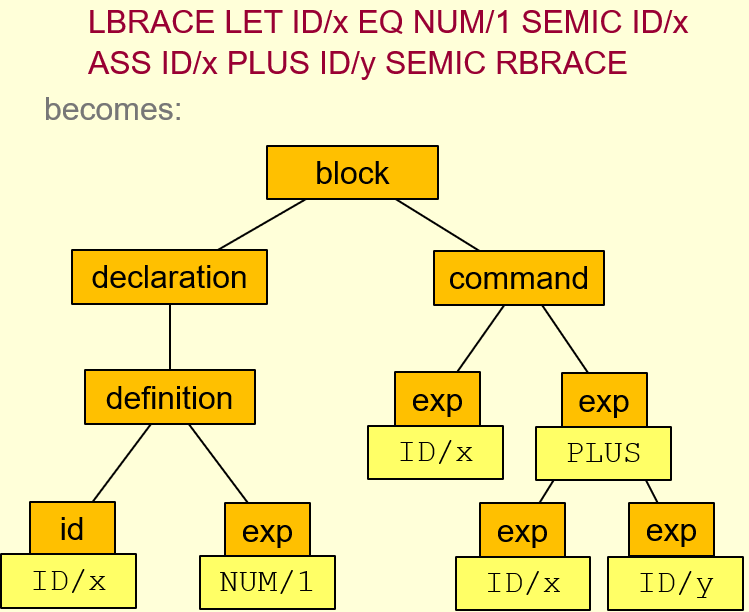
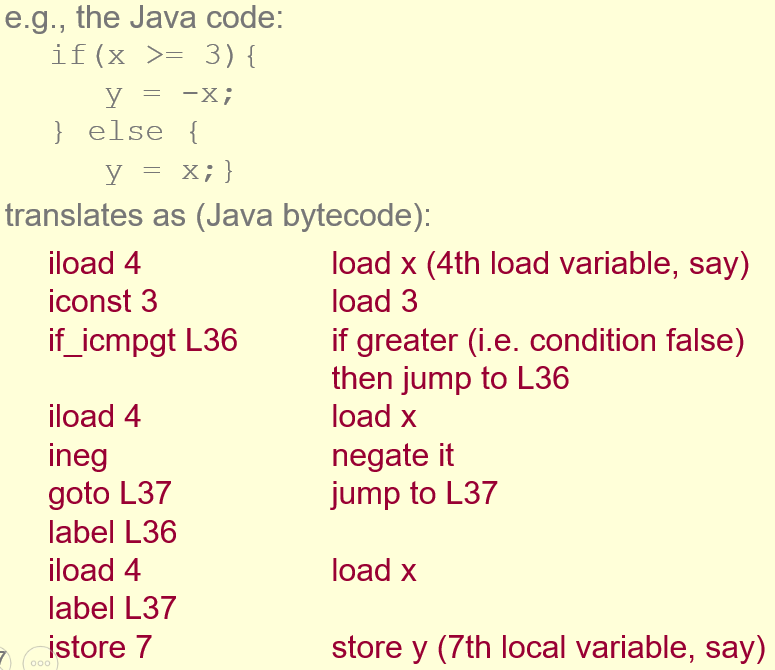
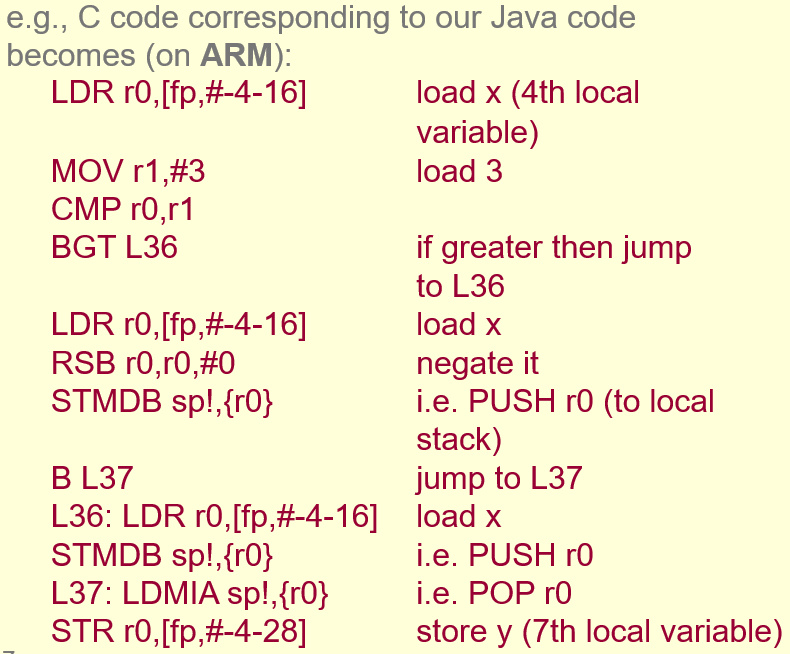
Programming languages should be easy to use, i.e. easy to read, write, understand and maintain, and support abstraction to add new features.

Every programming language has syntax – how it is written, and semantics – what a program means. Semantics should be formal, as informality leads to ambiguity which can cause errors.

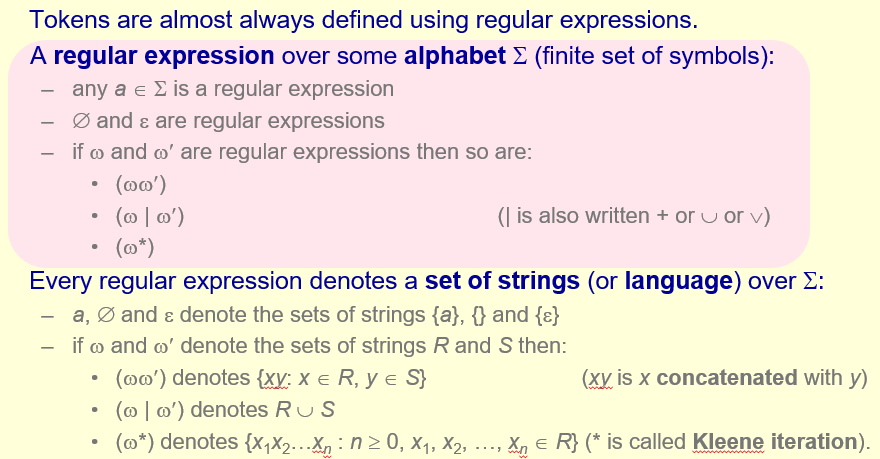
Computer science turns to maths (uh oh) to define formal semantics, e.g. set theory, category theory and logic ☹, e.g. program meanings can be given mathematically as a function (denotational semantics), in terms of the steps of a computation the program makes as it runs (operational semantics) or indirectly in terms of axioms (oh no).

Compilation vs Interpretation:

A compiler translates a high level program into machine code.

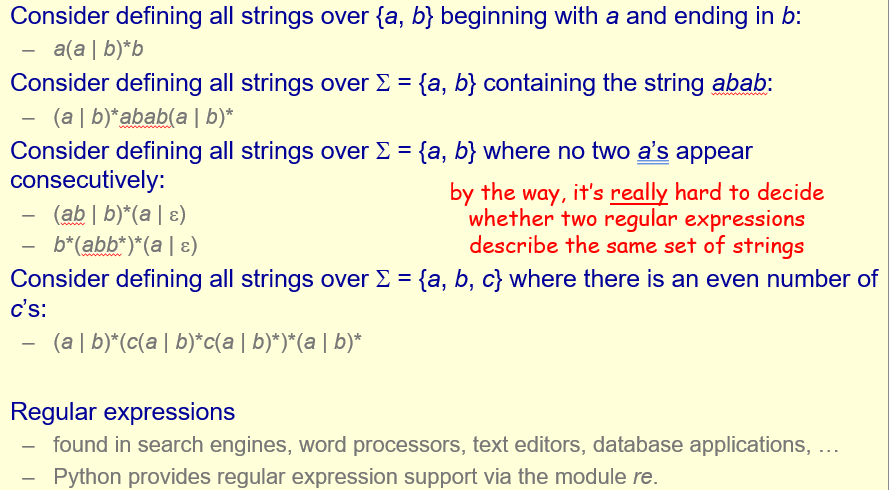
* Lex – Lexical analyser reads strings of symbols and converts them into basic syntactic components, i.e. tokens in a token stream.
* n – Syntax analyser or parser recognises the syntactic structure of the token stream and results in a parse or syntax tree.
* Trans – translation phase translates the parse tree into a linear sequence of intermediate code, e.g. bytecode.
* cg – code generation phase converts intermediate code into assembly.

Lexical analysis

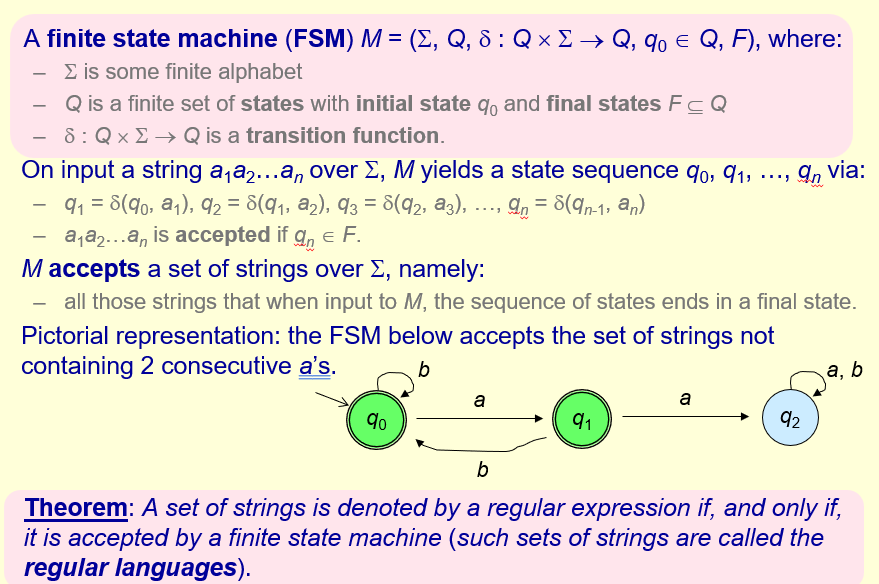
Lexical analysis can account for over 50% of compile time. Character handling is expensive and there are a large number of characters in a program.

(uh oh set notation )

Examples of regular expressions:



FSMs

(uh oh Greek letters)

For every FSM to define a language there is an equivalent regex ad vice versa.

**Efficiency and Complexity**

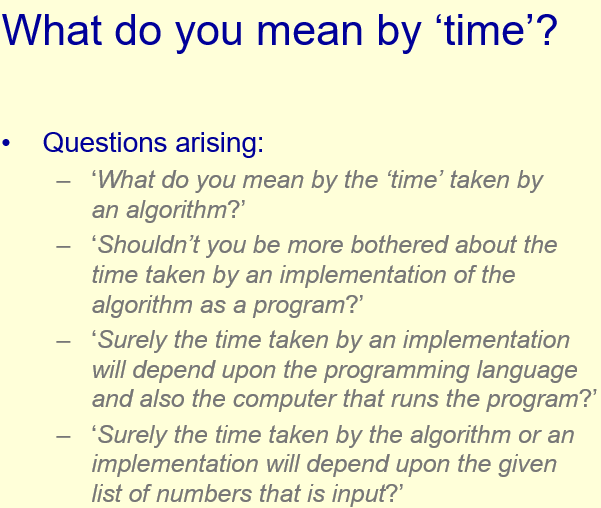
How do we measure how good an algorithm is?

We have to decide on a resource of interest, e.g

* Time taken
* Amount of memory used during the execution of the program.
* Ease of implementation of the algorithm
* Amount of memory used to store the program itself

Etc.

Our resource of interest is time.

^ we will “try” to deal with these questions – may not be to everyone’s liking.

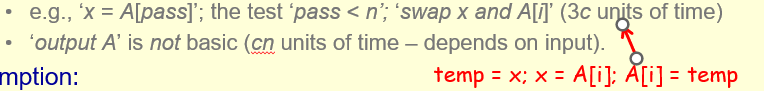
Analysis of algorithms – the consideration of algorithms from the point of view of how good they are, wrt to some resource (time).

If we focused on implementations, our analysis of time will vary based on the machine, programming language, compiler, processor etc used to implement the code. As a result we work only with the algorithms themselves to produce a machine-independent analysis that is relevant to any implementation of the algorithm on any given processor. We focus on improving the algorithms themselves, rather than tweaking an implementation of the algorithm to run faster on a specific processor.

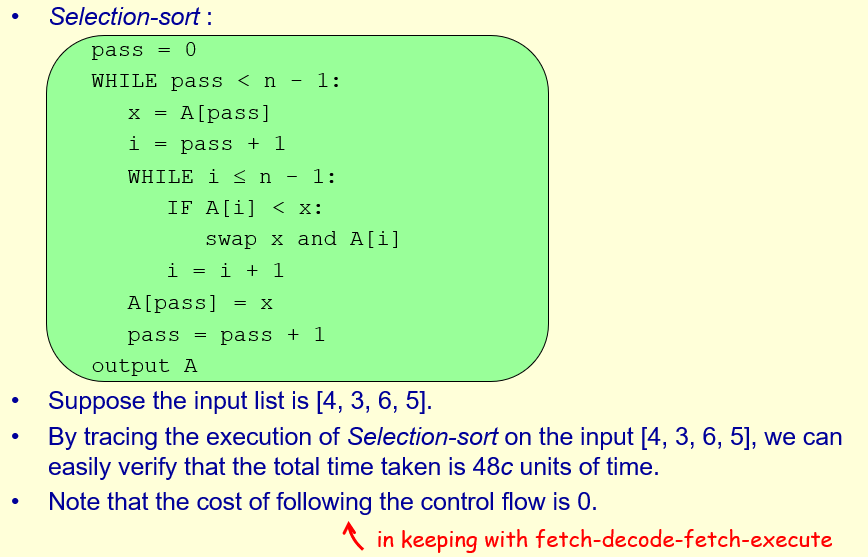
Problem – we use pseudocode to write an algorithm that is machine independent, but this has no formal definition, making it impossible to give a formal definition of the time taken by an algorithm.

Solution – Analysis is never precise, we hide and work around irrelevant details. We assume variables can be manipulated in a small constant, c, units of time, as can basic operations involving variables. Actual value of c depends on processor and implementation so we don’t care what c is.

More generally, any basic algorithmic operation can be undertaken in at most c units of time.

We also assume the input numbers are always small enough to be dealt with by any processor.

E.g. Selection sort

Problem – there are infinitely many inputs to selection sort, we have only calculated the time taken for one of them

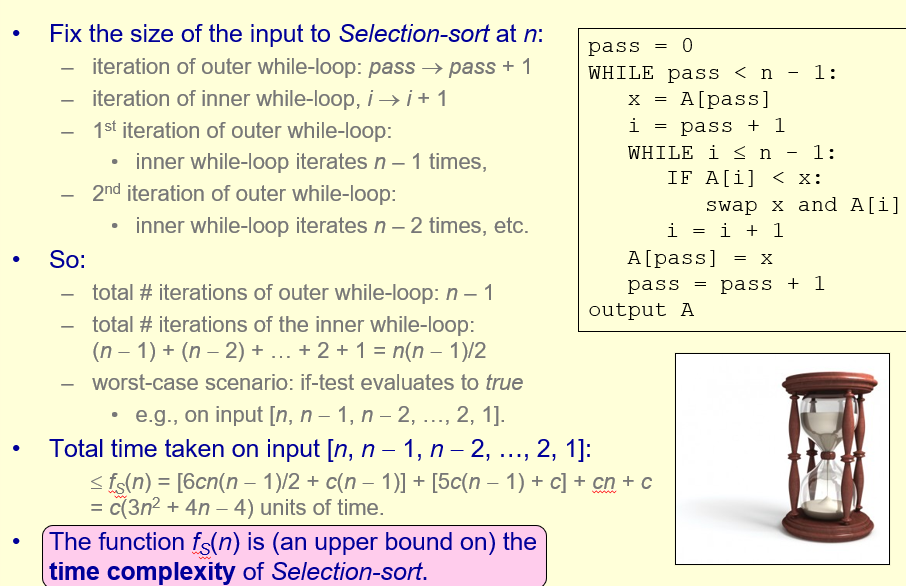
We would like a worst-case upper bound, so that regardless of input Selection sort terminates in b units of time. This is not possible, as a list of length b+1 it requires (b+1)c units of time just to output this list (final line).

As a result, we group inputs together based on their size.

We need to express our worst case upper bound as a function of f: N->N so that any input of size n is such that selection sort terminates in at most f(n) units of time.

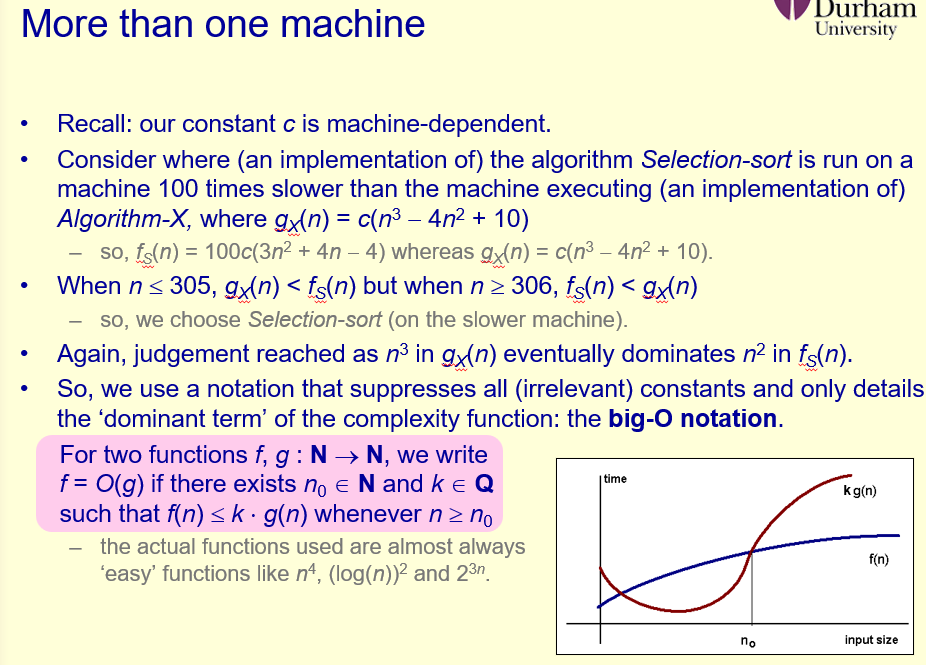
Assumption: The size of an input is the number of basic objects it contains (e.g a list is not one object), and that the time taken by the algorithm is strongly influenced by and increases with input size.

So we have reduced our problem to finding the worse case time for an input of size n, rather than overall.

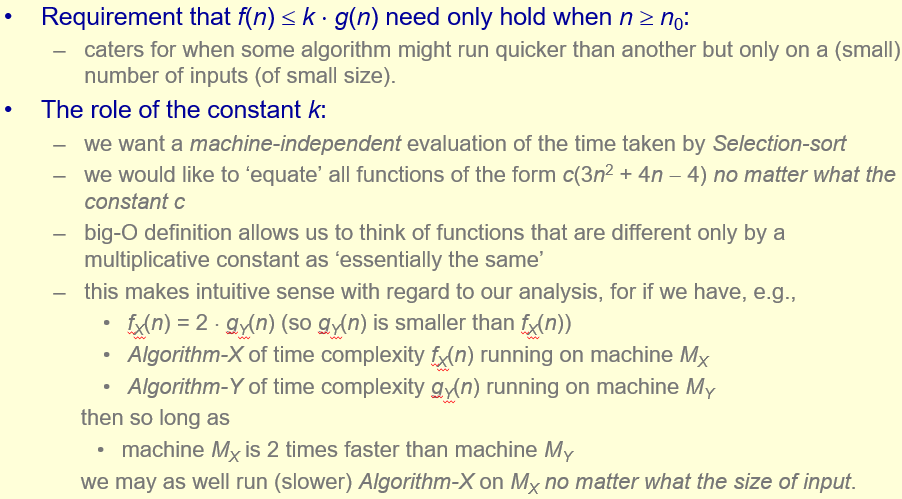
^the 6 in the first term comes from the 2 comparisons, the assignment and 3 for the swap. The cn is the outputting of the list and the c is assigning pass = 0

How do we use this time complexity formula? – To compare algorithms with the same functionality.

E.g. we can compare the time complexity formulas for different sorting algorithms.

Note – we do not care what happens for small inputs, only for large inputs 😊. Decisions made on which algorithm is better does not depend on the actual time complexity function, or the value of c; you only need to compare the dominant terms of two algorithms to compare their time complexity.

Reasoning behind Big-O:

Given Big-O, we need not take so much care when analysing algorithms, e.g for selection sort we have c(3n2+4n-4) which is O(n2)

Or, we can just say all instructions, except the output, take constant time, and that time complexity is dominated by the number of iterations of the outer while loop: n-1, and the number of iterations of the inner while loop: at most n-1 per iteration of the outer loop.

Hence, time complexity = (n-1)(n-1)+n = O(n2)

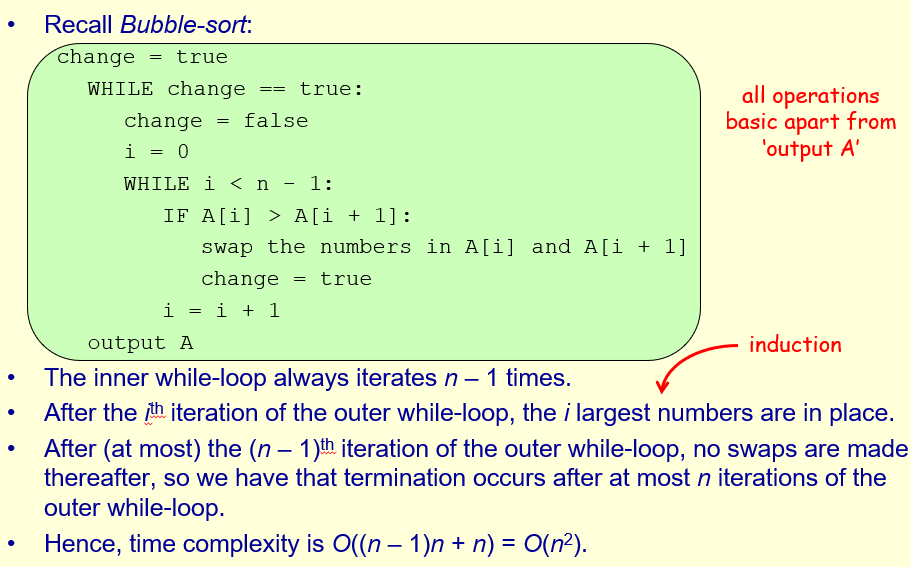
Identities:

2100n3 = O(n3)

34n3 = O(n5)

2034n789=O(2n)

aini + … + a0 = O(ni)



Practical relevance: asymptotic time complexity is usually a reasonable practical guide for comparing algorithms. However, hidden constants may be large enough to have a significant practical impact. Furthermore, worst-case analysis may not be realistic as some algorithms have few worst-case algorithms, or do not arise often in practice, e.g. Quicksort has worst case O(n2), but average case O(nlogn). However, average case time complexity is often much harder to calculate (beyond this module 😊)

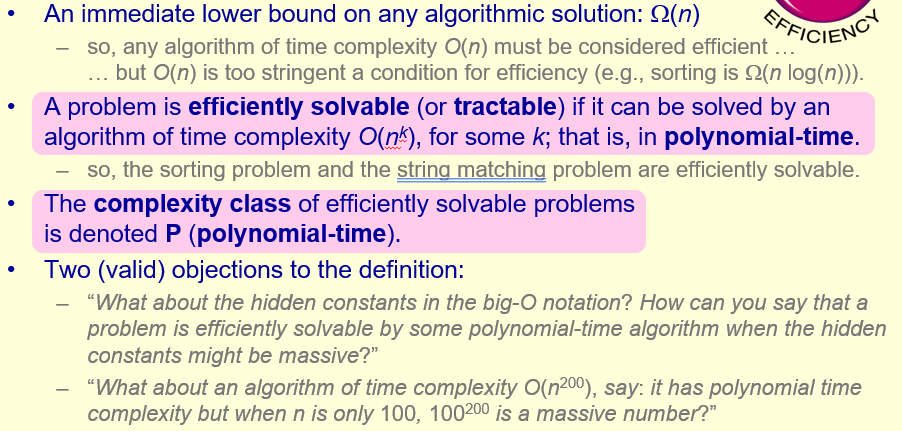
If memory is sparse, alternative algorithms such as Selection sort may be preferred even if they have worse time complexities, as they are “in-place” algorithms without needing much extra memory like Merge sort.

What exactly is a hard problem?

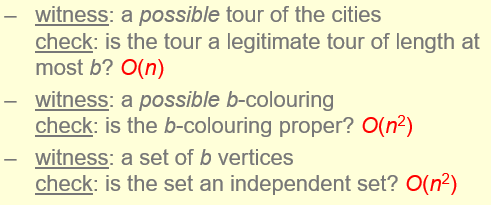
* Note a problem is a decision problem unless otherwise stated

We want our algorithms to provide answers on inputs of a reasonably large size, and it should provide these answers reasonably quickly.

What makes a problem efficiently solvable?

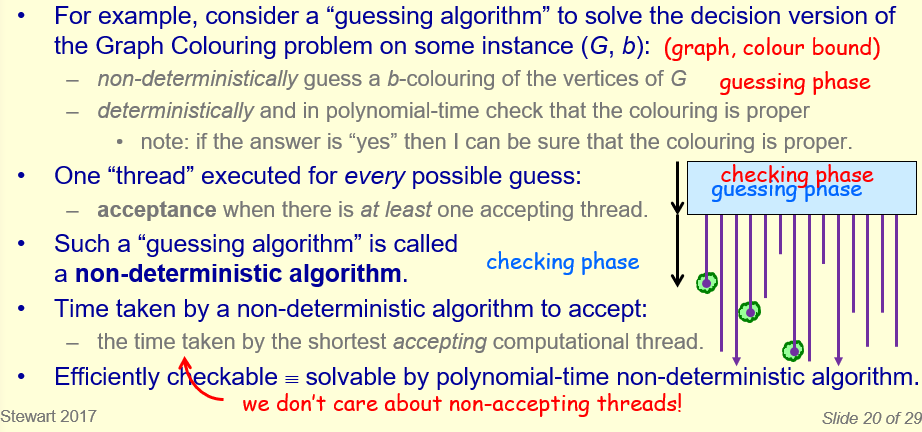
Many problems we would like to be efficiently solvable don’t appear to be , e.g.

* Decision version of TSP
* Decision version of Graph Colouring problem
* Decision version of the Independent Set problem

Do they have a common property?

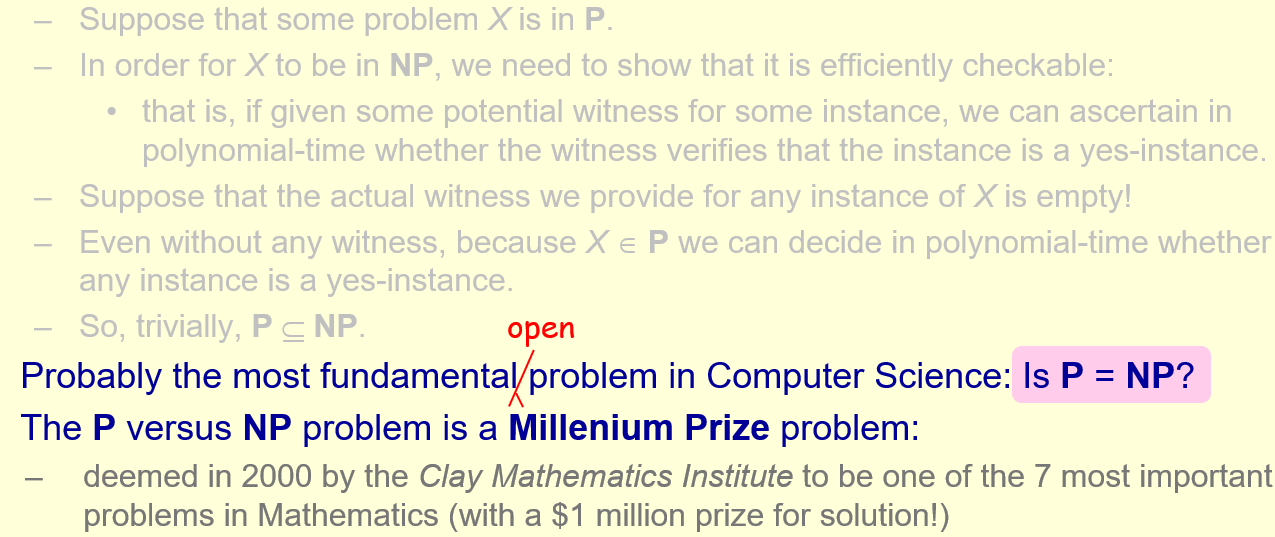
^witness = a possible solution you can check against

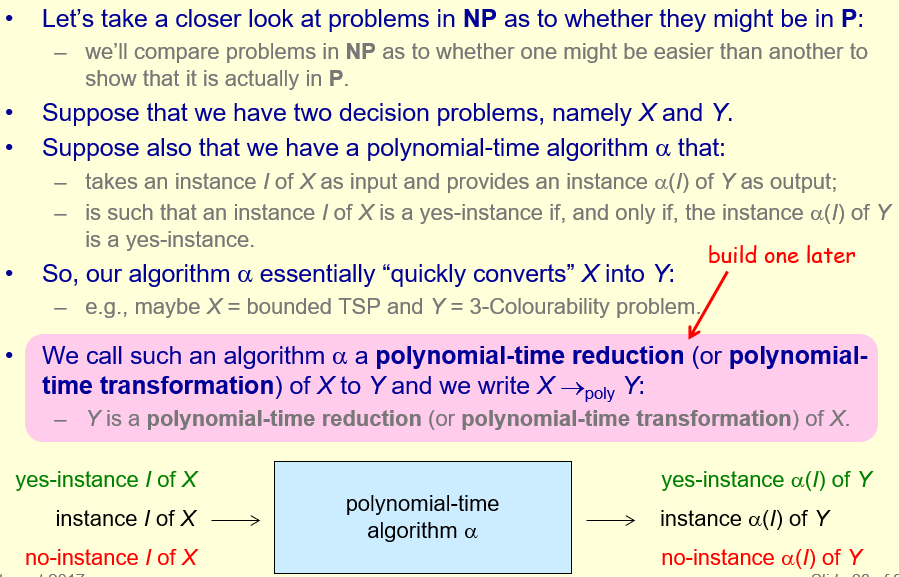
A decision problem is efficiently checkable if given a potential witness, we can check in polynomial time whether that witness is indeed a yes-instance of the problem.

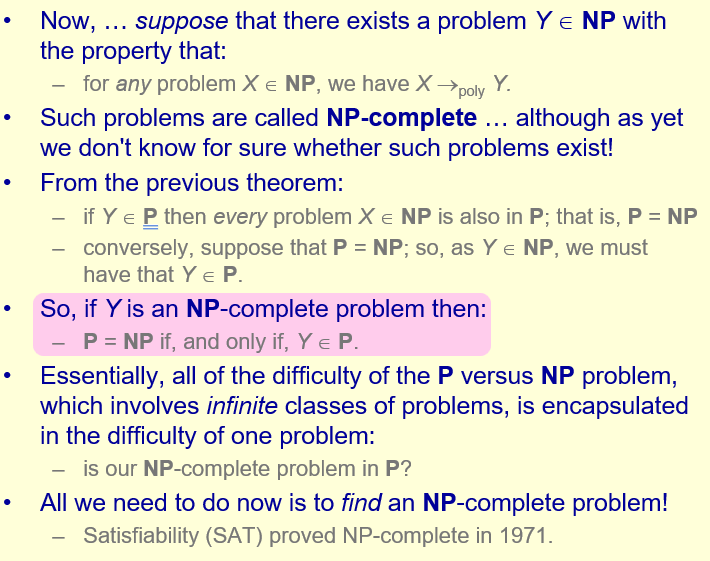
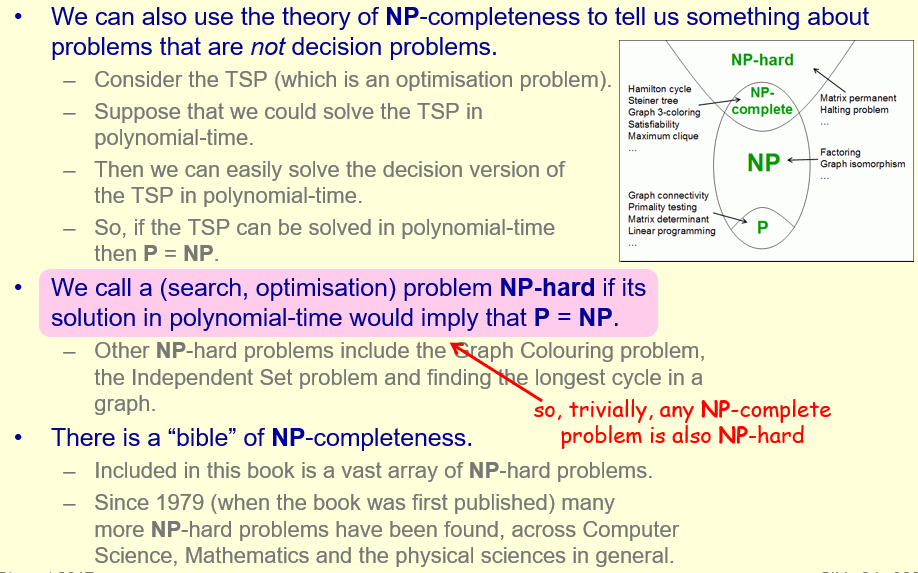
i.e. we are allowed to guess within a polynomial time algorithm to solve the problem.

Complexity class NP: (non-deterministic polynomial time) consists of those decision algorithms that are efficiently checkable. Many problems we *want* to be efficiently solvable, i.e. in P, are in NP.

How do P and NP relate?

^(uh oh)

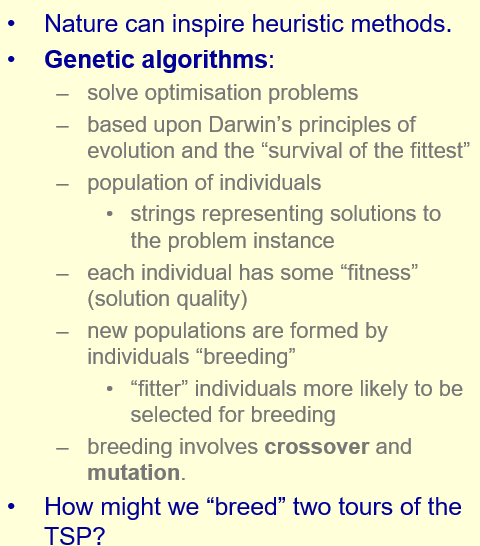
.

a

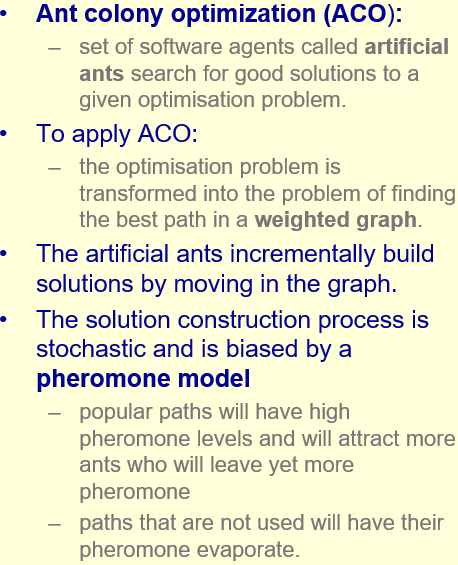
How can we solve hard problems?

* Brute force 😊 – ignore NP-hardness, generate all possible witnesses and pick the best. For example, the TSP for n cities has n! tours – not realistically solvable for large n (well even n = 29).

However, the best exact algorithm for the TSP known so far is brute force.

* Heuristic methods:

^crossover = take two solutions, randomly split them at a certain position and swap the sublist after this position for the two solutions. Mutation = randomly change a value within the solution.

* Ant Colony Optimisation (ACO):

^for large graphs it takes a long time for ants to narrow down to a specific path. Also need ants to survive long enough to explore the whole graph.