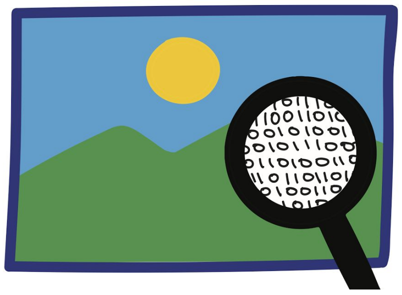
The Big Ideas of K-12 Computer Science Education

Prepared by:
Tim Bell, University of Canterbury, NZ
Paul Tymann, Rochester Institute of Technology, New York, USA
Amiram Yehudai, Tel Aviv University, Israel

When teaching computer science it can be easy to focus on details and lose sight of the bigger picture; this is particularly concerning with new pre-tertiary curricula being adopted in many countries, as teachers grapple with a bewildering array of topics to teach. Why do students need to know how to "code?" Why do we teach them how to work with binary numbers? What's the purpose of learning selection sort and quicksort? This document presents a list of 10 "big ideas" of computer science that have been distilled based on input from curriculum designers and computer science education experts around the world.

Although the 10 ideas have been numbered to help us talk about them this does not indicate an order.

1. Information is represented in digital form.



A huge variety of information is stored as data on digital devices, and shared between them; the data may be as simple as the number of steps counted on a fitness tracker, or as complex as the details of every transaction going through an international organisation; it includes text, images, video, sound and scientific readings. The remarkable thing is that all of this information is reduced to digits (that's the fundamental thing that makes digital devices so useful).

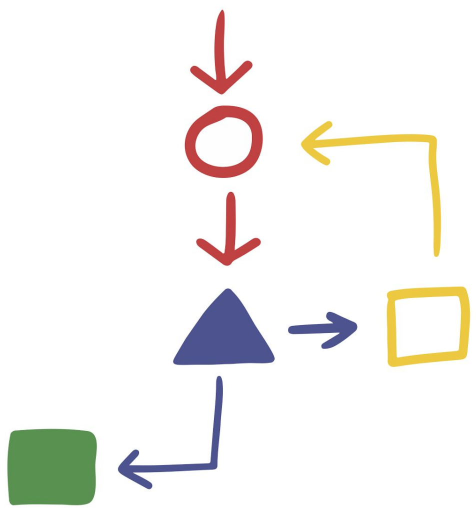
Digital representations lead to versatile devices because the same hardware can be used for quite different purposes: a smartphone can play music, take photos, send email and show videos, because all these things are represented as digits, which are easily stored, copied, manipulated and transmitted on the same hardware. This is in contrast to non-digital (i.e. analogue) devices, which by nature are specialised (phones connect to a phone line, a TV gets a signal from a TV aerial, music is played from a vinyl disc, and video is recorded on videotape). Digital data can also be shared without loss of quality, whereas analogue devices tend to reduce the quality if the material is copied or re-transmitted.

Digging deeper:

- The digits are usually represented in binary, although the key is that they are discrete (not continuous) values. Traditionally these binary digits are written as 0 and 1, but in practice they use electric, magnetic, audio and optical representations. The choice of just two different digits is essentially an engineering tradeoff of cost and complexity (it's easier to distinguish between two different values than, say, 10 different levels), but it relates to the idea that the simplest possible number system is binary (Claude Shannon pointed this out, and suggested that "bit" might also stand for Basic Indissoluble uniT). The basic circuitry that manipulates the bits is based on Boolean logic, and the simplicity of having only two values makes reliable electronic circuitry relatively inexpensive to construct.
- Despite a bit being so simple, you need relatively few digits (bits) to represent very large numbers e.g. you can write a number larger than the number of atoms in the universe in using a few hundred bits.
- Although the digits are represented physically, the scale is very different to what we are used to in the physical world; a Blu-Ray disk holds about 200 billion binary digits in a few square centimeters, and a Fibre-optic cable can transmit a similar number of bits in seconds, at the speed of light. This means that it is feasible to store and share useful amounts of all kinds of data on mobile or desktop devices, and for mass storage there are also very large general purpose systems (servers and cloud storage) that store data for multiple users and purposes.
- Traditionally data is converted to a series of **b**inary digi**ts** (bits) and represented using electrical, magnetic, audio, or optical representations. Different approaches to representing data as bits can improve how efficiently and effectively it can be stored, transmitted, accessed, and manipulated. For example, compression methods are used to reduce the size of a file, and encryption can be used to prevent others from discovering the contents of the file, whether they are intruding onto your computer, or eavesdropping on data that you are sending through a wired or wireless network. Nevertheless, compressed and encrypted data is still stored as digits.
- Techniques for manipulating information are at the core of the discipline, and hence processing digital representations is fundamental, whether it is changing the brightness of a photo or adding up the bill on a cash register.
- Within a programming language the representations of stored information is articulated as *data types* (e.g. integers, characters, strings of characters, and composite types that lead to more complex structures). They are all bits, but the programmer can decide that the purpose of the bits are in each situation. This also applies to files; in a text file the bits are interpreted as textual characters, while in a spreadsheet file most of the bits may be interpreted as numbers, and in an image they might be interpreted as representing colours.
- Programs themselves are stored as data; the same hardware that stores music or video is also used to store the app (software) that can display those files.

- Values in the physical world are *analogue*, which means that there are no discrete steps in values (e.g. for any distance or time, there is always a distance or time that is a little smaller, setting aside questions around quantum mechanics!). In contrast, digital devices represent the physical world as discrete values (such as treating a photo as a few million discrete pixels, or a sound as sound pressures selected from a limited range of values). The digitisation of analogue measurements is an example of abstracting away detail for the sake of simplifying the representation. The more detail that is removed, the simpler it is to store and process, but eventually the abstract becomes an oversimplification. The goal is to remove detail that is irrelevant to the computation or beyond human perception, to avoid using too many bits to store detail that won't be needed. The digital representation will nevertheless not be an exact version of the original.
- Computers move digits (bits) around through various hierarchies of memory, from very fast cache memory to off-site data warehouses.
- Digital representations must be invented for each type of data we represent. Creating new representations (such as new alphabets or languages) gives us the opportunity to think about how new writing methods are developed, and how they can be designed for different situations, giving some insight into the way that languages and alphabets have developed over the history of human-kind.
- Storing data requires physical matter (such as transistors), and transmitting it requires energy (such as light over a fibre optic cable). This can be done very efficiently in modern systems, but limitations in data storage capacity and transmission speeds exist because of the physical process involved.
- Computers exist that use a model other than simple digital representation (e.g. analogue computers and quantum computers), but currently digital devices (primarily using binary-based representations) dominate the devices that we use because this is currently the best engineering tradeoff for the various desired attributes such as economy, power consumption, size, reliability and versatility.

2. Algorithms interact with data to solve computational problems.



An algorithm is a well defined process that acts on data to solve some problem i.e. to achieve a result, such as finding the shortest route on a map, matching two strands of DNA, or changing the brightness of a photo.

An algorithm can only include steps that a conventional computer could do; for example, you couldn't just put in a step that says "find the most efficient solution". Remarkably, the full power of a conventional digital device can be realised by an algorithm using just three structures to control program flow: sequencing (putting instructions one after the other), selection (choosing which part of the algorithm to execute based on some values, usually using an "if" statement), and iteration (repeating part of the algorithm with a loop). Apart from these three basic types of instruction, a computer is able to read in information (input), give out information (output) and store data to use later on. These basic components can be used to define every algorithm, as they define exactly what can (and can't) be done on conventional devices.

Digging deeper:

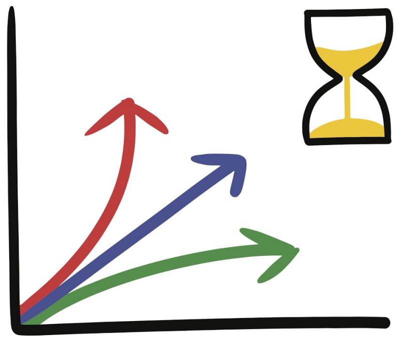
- Algorithms can be expressed as a static (finite) representation that describes a dynamic (potentially infinite) process.
- The "conventional" computer referred to here is the kind of processor found in typical digital devices, including personal computers, smartphones, supercomputers, cloud services, internet devices, and digital watches. Examples of *un*conventional computers would be quantum computers and analogue computers.
- The kind of algorithm that can be used in a given situation depends on the way that data is stored and organised (the organisation is referred to as data structures). For example, finding the smallest file in a list sorted in ascending order of size has a simple algorithm (just pick the first file!), while finding the smallest file in a huge disorganised list requires a different algorithm. There is a close relationship between algorithms and data; designing computer systems often involves tackling the tradeoff between efficient data storage and efficient algorithms to process the data, so it is therefore very important to be aware of the advantages and disadvantages of different ways of storing data.
- The term "computational problem", "algorithmic problem", or simply "problem" in this context is often used to refer to the task that needs to be computed e.g. searching for a word, sorting values into order, finding the shortest route on a map, or finding a face in a photo. These kinds of problems are different from, say, a maths problem, where students might be expected to find a single correct solution. A computational problem can have several correct solutions (algorithms), and an algorithm is a general process for solving that type of problem. For example, if the problem is how to search for any given word in any given document, there are several possible algorithms (e.g. a sequential search comparing every word; or sorting the words into alphabetical order and then searching the alphabetical list); these computational solutions contrast with a specific outcome, such as "where is the word 'score' in the Gettysburg address?", which has the simple answer that it is the second word.
- Algorithms should solve for all possible inputs that might be given in the future (such as finding the high score in *any* list of scores); this contrasts with other human problem solving that solves for just one instance at a time e.g. design a bridge for a particular river crossing.
- The model for a conventional device (using sequence, selection, iteration, input, output and storage) goes back to the first electronic computers, and is based on work done by Alan Turing and Alonzo Church; conventional devices and programming languages that meet these criteria are often (loosely) referred to as "Turing-complete". Because all conventional digital devices that we work with meet these criteria, it means that designing algorithms using this model has very broad applicability. *The concept is*

explored further by Guzdial here https://computinged.wordpress.com/2012/05/24/defining-what-does-it-mean-to-understand-computing/. Turing's model can also be expressed using an equivalent approach called recursion, which can make an algorithm easier to reason about how it works.

- Algorithm correctness is important - we need to be sure that it achieves exactly what we expect it to achieve. This includes heuristics (near enough solutions that may not be perfect, but can be computed faster than the optimal solution); heuristics can be considered to be the correct solution if the desire is for a solution that is fast enough and near enough to the best possible solution.
- The interaction of algorithms with data is crucial; we can choose between various ways to arrange data in computer storage (through "data structures"), which affect how quickly it can be accessed. Whenever the form of data is changed, there is typically a tradeoff between time and space e.g. a search data structure such as a "hash table" is faster than an unstructured list, but wastes space. Data structures can give faster access to data for purposes like searching for information, finding patterns, and calculating paths through maps. There are many kinds of data structures used on computers, ranging from variations on lists, to trees and "graphs", which can represent general networks such as road maps. The relationship between data structures and algorithms is seen, for example, between searching and lists; a sorted list can be searched more efficiently than an unsorted list because it makes a better algorithm possible, but it requires more computation in advance to ensure the list is in sorted order.
- The data can also have the fundamental bit-level structure of how it is stored changed to make it take less space (compression), to make it meaningless to eavesdroppers (encryption) or to make it useful even if a few of the bits get messed up (error control). This also presents tradeoffs; for example, a JPEG file is smaller than a raw image file, but requires processing to make it smaller, and there will be a tradeoff in quality. The technical word for changing the bit-level representation of data is "coding", although the meaning in this context is quite different to the meaning of "coding" in the context of programming. Coding changes the form of representation for encryption, compression or error control to better suit how we wish to use it.
    - Encryption can allow us to transmit data while someone observes all transmissions (including setting up the encryption keys), yet the eavesdropper is still unable to decode anything from the data. This enables a secure connection to be set up with a bank or online shop.
    - Compression has the potential to reduce the size of files so they use less storage and are transmitted faster; some methods however can expand the files rather than compress them, and it is impossible to achieve infinite compression. Compression is particularly important for large media files such as photos (e.g. JPEG, GIF, PNG), video (e.g. MPEG) and audio (e.g. MP3).
    - Error control enables us to detect when data has been corrupted, to any level of confidence required. All data stored long-term on disks and flash memory, and all data sent over wireless, wired and fibre networks has error control added to it to make sure that the user doesn't accidentally end up working with incorrect data.

- Many algorithms rely on decomposition using "divide and conquer" techniques, where the problem being solved is being reduced in size until it is small enough to deal with easily (e.g. sorting one item into alphabetical order is trivial!) A key idea here is recursion - expressing a problem as the combination of smaller problems, which themselves can be broken into smaller problems, and so on until you reach a trivial "base case". The problem might have millions of items in it to process (such as searching for a word in a list of a billion words), but the base case typically has only one, or even zero, items in it (such as searching for a word in a list of just one word).

3. The performance of algorithms can be modelled and evaluated



The main resources that an algorithm uses are time and space (memory). Time is a key factor because slow programs are annoying to users, and if a program is going to take decades to complete a calculation, it's better to work that out before you go to the trouble of implementing it! Using an unnecessarily inefficient algorithm will also lead to devices wasting power or needing cooling, which have an environmental impact; or it could make the battery on a device go flat too quickly. Some algorithms also need a lot of spare memory or storage while they are running. This may make the algorithm infeasible in some cases, while in other cases it might be an excellent tradeoff if the algorithm is faster.

The time taken to solve a problem with an algorithm (and therefore programs that run the algorithm) isn't necessarily proportional to the size of the input; sometimes it's better than that, and sometimes it's a (lot) worse. The time taken by an algorithm is usually estimated based on the size of the input (such as the number of items being searched through, the number of streets in a map, or the number of pixels in an image). It's important to at least estimate the speed of an algorithm before implementing it, as it might be very sensitive to the size of the input; perhaps a program works satisfactorily in tests, but with a larger input it might take a *lot* longer.

Digging deeper:

- There can be many different algorithms for solving the same problem, but some are more efficient than others (for example, you could look for a book in a library by starting at the first shelf and checking every book, but it's better to take advantage of the order that the books have been shelved).
- The time taken by an algorithm on particular input is often measured as a function of the amount of input to the algorithm. Sometimes the time taken by an algorithm is proportional to the amount of data (given twice as much data, such an algorithm would take roughly twice as long to process it), but very often the time taken is *not* proportional. Many algorithms take more time than would be predicted from an assumption of taking

proportional time (for example, sorting algorithms generally take more than twice as long to sort twice as much data); there are also some algorithms (such as binary search and hashing) that hardly take any extra time even to process a problem that is 10 times as big. Some algorithms become completely infeasible if even a little more data is added to the problem because they require exponential time in the amount of data they are processing; for example, there are algorithms where adding one extra item of data can double the processing time. Such problems that don't have feasible algorithms are referred to as "intractable" (see the next big idea).

- The rate of growth of the resources needed by algorithm is referred to as its *complexity.* Complexity measures are not usually made precisely, since the time taken will depend on the particular computer and other details, so instead they consider the rate of growth; for example, the notation O(n) is used to indicate that an algorithm solving a problem of size n takes and amount of time proportional to n, whereas $O(n^2)$ indicates that it will take time proportional to the square of the amount of input (doubling the amount of input will take approximately 4 times as long to process).
- Choosing the wrong algorithm for a situation can lead to unnecessary computation, which uses power (e.g. battery life on a portable device, or expensive energy in a large data centre); this in turn can have environmental impacts.
- It is possible to quantify a lower bound on the amount of time that a problem may take to solve even if we don't have an algorithm in mind; as a trivial example, for most problems, if a program is processing *n* values, then it at least needs to take the time to read in all *n* values. An algorithm gives an upper bound for how long the problem will take to solve (since it solves the problem, but we might not yet know if a faster algorithm exists). For some problems we have a large gap between the known upper and lower bounds, while for others we know what the best possible algorithm is.
- Computer science routinely deals with very large and very small quantities. Online systems can deal with billions of customers or transactions, cheap cameras capture millions of pixels in a fraction of a second, personal computers store billions of binary digits, data usually travels at the speed of light, and a step in a computer instruction happens in a billionth of a second, yet some algorithms can take billions of years to complete. It is important to be able to evaluate these situations, as sometimes they work in our favour (e.g. a code that takes billions of years to crack) and sometimes against us (e.g. an image enhancement algorithm that takes hours to complete).

4. Some computational problems cannot be solved by algorithms.

There are some computational problems that we can prove will never have programs written to solve them (these problems are not computable). For example, we can prove that no one will ever be able to write a general app that can vet another app to tell you if it will freeze your smartphone (more formally known as the halting problem).

In addition to non-computable problems, there are many practical problems for which all known algorithms to find optimal solution are “intractable”, which means that no machine exists that has the resources required to execute the algorithm once the size of the input gets fairly large. For these problems we need to consider heuristics (that is, find an approximate solution) rather than pursue optimal solutions that could potentially take billions of years to evaluate on the fastest computer. Some problems have mathematical proofs that they are intractable, but there are many problems for which we haven't found an algorithm that runs in a reasonable amount of time, despite decades of research; yet we also haven't proved that the algorithm can't exist. Resolving this issue is widely regarded as one of the biggest questions in computer science!

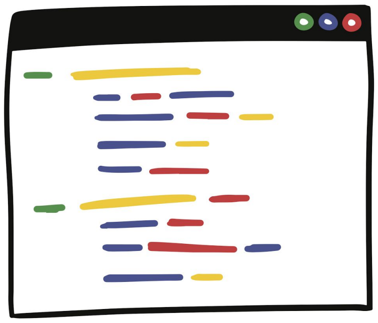
Digging deeper

- A consequence of these ideas is that the idea that "computers can calculate anything given the right program" is incorrect. The limits of what can be computed may change if new types of computing become common (e.g. quantum computing), but with the devices that have been in use since the dawn of electronic computers, there are definitely things that can't be done.
- Another example of a non-computable problem is the "line of code" problem. If you write a large program, you might want to know for a particular line of code in the program if it ever gets executed, but no-one can ever write a program that reads in another program and tells you if a particular line will be executed. Of course, it can be done in some special cases, but no general program can be written to achieve this. Another program that can't be written is one that is given two other programs, and decides if they always produce exactly the same output if they are given the same input.
- Problems for which we have only algorithms that take an exponential amount of time (or space) as the size of the input increases are generally regarded as *intractable* as the time needed can blow out to billions of years very quickly. Such algorithms amount to trying out every possible answer to find the best. (Exponential time typically means that to process *n* items of input, the time taken will be proportional to 2^n or worse, which doubles each time just one item gets added. This grows extremely large very quickly - for example, adding 10 more items will multiply the time by more than 1000.) There are examples where this is frustrating (e.g. finding the perfect route for a courier to drop off parcels is intractable; we can find routes that are probably perfect or nearly perfect, but it would take too long to be sure we have the best possible route, and therefore we may be wasting fuel and time); on the other hand, this negative situation applies to cryptography, where cracking the codes that are in widespread use is intractable, and the lack of algorithms to do this makes secure communication and commerce possible online.
- For some problems we know the optimal (fastest) algorithm for solving them because we can prove it can't be done in fewer steps, but for others we haven't discovered tractable algorithms yet, and in some cases we don't even know whether or not tractable algorithms exist. One of the biggest open questions in computer science, called the P vs. NP problem, is related to intractable problems. There are two interesting variations of

intractable problems: design an algorithm that produces a solution to the problem, or design an algorithm to check that a given solution is correct. For this huge family of thousands of important problems (e.g. timetabling, route planning, vehicle loading, and DNA matching), we have efficient algorithms for checking if a given solution is correct, but we do not know if an efficient algorithm for finding a solution will ever be discovered. As an example, consider the bin packing problem, in which objects of different sizes have to be packed into a finite number of containers, each of a fixed capacity. If we are given a proposed packing (which items are to be packed into each bin), checking that this is a valid packing is an easy task that can be done efficiently - you just need to check that no container is too full. But we do not know of any efficient algorithm that will always decide if the given objects can be packed in a given number of bins. The remarkable fact is that if an efficient algorithm is found for any one of these problems, then all the other problems will also have an efficient algorithm, and similarly if we show that no efficient algorithm exists for any one of them, than all of them are intractable.

- Despite not being able to find efficient algorithms to find optimal solutions, many heuristics (algorithms that give near-optimal solutions) have been developed, and some are known to come very close to the optimal solution, which means that in many practical situations it is better to focus on improving heuristics, and not worry too much that the result might be very slightly short of optimal.

5. Programs express algorithms and data in a form that can be implemented on a computer



Programming involves taking algorithms (which might exist only in the programmer's head, or may have been designed by a team of people) and turning them into program instructions that can be executed by a computer. Program instructions are written in a programming language which is precisely defined. These instructions manipulate data on the computer, so the form and meaning of the data is dictated by the program.

Because the capabilities needed to fully control a general purpose computer (which covers most digital devices) can be defined by six properties, consisting of three control structures (often expressed as sequence, selection, iteration), and three ways to deal with data (input, output and storage), these properties are also the key elements of writing programs. Consequently, any programming language that has all of these elements can be used to write any computation that any other full programming language could be used for, and the differences between languages is largely to do with how well they suit a particular situation (e.g. for processing files, running on a smartphone, teaching programming, or running an enterprise system).

Digging deeper:

- The six properties are the same as the structures used to define algorithms:

    - Sequence: specifying instructions to happen one after the other
    - Selection: choosing between two different sets of instructions (commonly an "if" statement)
    - Iteration: repeating a set of instructions (loops)
    - Input: reading in data from outside the program, which could be from a keyboard, pointing device, a file, a network, or even another program
    - Output: getting data out of the program, which could be to a screen, through a network, into a file, or to another program
    - Storage: also known as memory; being able to store data for use later in the program (typically in variables) or long-term storage such as files on a disk.
- This means that if a student has learned enough programming to include the six properties needed, then in principle they have the tools to program anything that can be computed. A consequence is that even "simple" languages like Scratch are actually just as capable as the most advanced languages if the programmer has learned the right parts of the language; they just might take more work to achieve the same thing, and have less suitable input and output formats.
- The six properties derive from the idea of a universal computing machine articulated by Alan Turing, and computers and languages with this capability are often (slightly loosely) referred to as a Turing-complete system. The modern digital devices that we work with (from small portable devices, to large "cloud" computing systems) and commonly used language both meet these requirements, and are no more powerful than Turing's universal machine, so they are fundamental in defining what computers can and can't do.
- There are always different ways to implement a program and store the data it works with; this means that programming is an act of creativity, and there is no single way of implementing a program for a particular situation.
- Getting programs to work correctly isn't just a matter of "coding" them; developing programs involves designing the structures, testing them and debugging the software, and these are all important skills for a programmer to have.
- A programming language (such as Python, Scratch, Java, JavaScript or Basic) is designed for a particular purpose, and has a formal description of what can and can't appear in a program (syntax), as well as what each command does (semantics). Defining these rules is the domain of *formal languages*, which provides tools for concisely specifying how a given programming language works.
- Although all current devices and languages are (essentially) Turing-complete, there are more powerful models of computation, such as non-deterministic systems and quantum computing, but building them is challenging, and commonly used devices remain Turing complete (so far!).
- There are also less powerful models of computation (such as Regular Languages and grammars) that are easier to use and do the job for some purposes; these are sometimes used as a part of a program for doing some simple tasks such as checking the format of input values, and also checking for some of the syntax errors in computer programs!

- Computer programs are stored on computers in the same way as data is, represented using these zeros and ones, including the "source" code that humans write, and the machine code that actually runs on a digital device. This means that programs are also stored as data, and that computer programs are used to manipulate other computer programs in the same way that they manipulate data; the most common form of this is compilers and interpreters that allow us to write programs in high level language and have them run on a computer that uses a low level language. It also means that a digital device is very flexible; the program is just data that can be updated or replaced easily, so the same device can be used as a video player, a burglar alarm, a medical monitor, and a game system.

6. Digital systems are designed by humans to serve human needs.

This is the driver for all the ideas above; digital devices must be fast, reliable and match a need appropriately if people are to use them, and because they are designed by people, the process for designing them needs to enable the developers to efficiently turn creative ideas into working products.

This means that there are (at least) three broad areas concerned with human factors:

- *human computer interaction*, which is about creating interfaces that are easy to use in the situation they are intended for,
- *software engineering*, which is concerned with enabling organisations to develop software on a large scale (potentially with thousands of people contributing to it), and at the same time making sure that the product meets the needs of the user, is reliable, does what is intended, and is completed in a timely fashion, and
- *ethics*, which considers the impact of a new technology on humans, the responsibilities of those who work on it, and possibly even whether or not we should construct it.

All of these concerns require some understanding of human behavior (psychology), interaction (sociology), and capability (physiology).

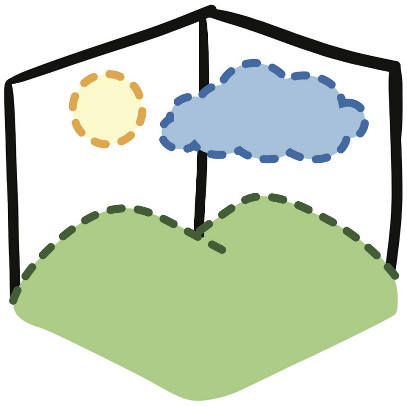
Digging deeper:

- Frustrating and inaccessable user interfaces are far too common, and "easy to use" systems that delight users sell for a premium. Designing good interfaces builds on all the big ideas in this document to make software work quickly and reliably, and also builds on the human sciences to make sure the final product matches the way people work.

- Although it might seem that programs are written for machines, the *text* of programs themselves is written by humans, for humans - most programs will be read by someone else later, and they need to be written in a way that they can be understood quickly and unambiguously (e.g. with meaningful variable names and comments). Knuth's "Literate programming" is based on this idea: "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do." (Knuth, 1984); and Abelson & Sussman (1996) wrote: "Programs must be written for people to read, and only incidentally for machines to execute."
- This also means that we highly value processes that enable people to collaborate efficiently and effectively in teams to produce software that works well for the end users; this is the realm of software engineering, and involves many techniques around working on large projects in teams to produce quality software.
- The processes used to build digital systems need to assure us that the product will work appropriately. This involves human processes itself; ensuring that a program is reliable requires designing thorough tests for it, and tracking down any bugs can require a lot of skill. Even then, a test might show the presence of bugs, but if software passes all tests, that doesn't necessarily mean that it is bug-free (this was articulated by E.W. Dijkstra).
- Decomposition is a general tool in computer science that breaks a large task or object into smaller components that can be dealt with without the distraction of the whole being visible while working on a small part. Building large software systems (software engineering) relies heavily on decomposing the task, as a large system needs to be built by multiple people working on parts of a system. This is sometimes seen as a "stack" of sub-systems; for example, communication systems have layers from the physical specifications through to protocols the programs can interface directly to regardless of the network being used (such as file transfer and email); web programmers use a stack of systems that simplify programming different elements of the website; and operating systems provide functionality through "drivers" that interface with specific hardware in a way that saves each software developer being concerned with the details of the particular computer being used.
- In a digital society our privacy and right to participate is often mediated by digital devices; furthermore, there may be some things that are possible to build that shouldn't be built, and informed discussion is needed, in the same way that discussion can be had around nuclear power, genetic modification or climate change if the public have a general understanding of the science. Ethical considerations mean that we should question algorithms that affect our lives (e.g. voting, privacy, job loss), the ownership of intellectual property, and the impact of digital systems on the environment (both positive and negative).
- The human aspect of computing is also reflected in how it appears in so many disciplines - computational biology and computational linguistics are just two

examples of how ideas central to computing are enabling new approaches in other disciplines.

7. Digital systems create virtual representations of natural and artificial phenomena.



Computer simulations and virtual systems are used to create virtual versions of *processes* in the physical world, and also to create imagined scenarios. Simulations can be used to reduce cost (e.g. simulating a structure to fine tune it before building it, or simulating different financial scenarios to choose the most effective strategy) and to reduce risk (e.g. simulating dangerous situations to give aircraft pilots experience, or simulating the spread of a disease to determine how best to prepare for an epidemic). Virtual *worlds* can be created by generating images and sounds artificially, to make the user feel that they are in a world imagined by the designer (including computer games, virtual reality and augmented reality). Virtual *machines* provide a simulation of a computer, and this approach is widely used because it protects the physical hardware and the software from each other, which can provide a safer and more flexible environment. Artificial Intelligence models human intelligence, in an attempt to replicate human decision making and reasoning.

These virtual representations take advantage of the unique properties of digital devices that enable a system to work on vast amounts of data, and if something goes wrong, they can be re-started in full working order by simply restoring some data, which is considerably simpler than reinstating physical objects that have been damaged or placing humans in a dangerous situation.

Digging deeper:

- The virtual representation of a real-world phenomenon is often referred to as a *model* since it is an approximate, but useful, version of the original scenario; this could be a model of the forces on a physical structure, a model of a financial scenario, or a model of the objects in a 3-dimensional image being viewed on a screen or a virtual headset.
- Simplified models can be used because some details will make (almost) no difference to the solution to a problem; abstracting out those details can save time and space, and avoid having to capture detail for no material purpose.
- Simulation can be used in science where empirical or theoretical approaches won't work. Simulation has become a "third paradigm" of science, joining experiment and theory as an approach to discovering new knowledge.
- Games and virtual worlds create a new environment that may never exist in the physical world (such as a fantasy game or animated movie), or they may seek to model

something in the real world as accurately as possible (such as a flight simulator or the spread of disease).

- Computer graphics and visualisation systems enable us to project 3-dimensional models onto a computer display.
- Computer systems also model the world in real time - for example, databases and spreadsheets are used to track the amount of money in a bank account or the score of a student; such models abstract key ideas from the world.
- Artificial intelligence is still a computer program, and so subject to the rules and limitations of computer programs (for example, they may contain errors); however, they may be sufficiently effective that they no longer appear to work as a conventional program.
- Large, detailed models (big data) can be used to search for patterns (data mining/machine learning) that may not be obvious in the physical world.
- Queuing systems are a common basis of simulation, where the system tracks entities (such as data on a network, or customers in a store) by considering their activities to be a series of queues, waiting for various services.
- Although we are modeling the "physical world" with simulations, a computer is still a physical device, and the data and programs are stored physically. However, they are on a completely different scale, and are stored in a way that is very flexible - a flight simulator can run on the same physical devices as a disease simulation or a movie animation.
- A virtual machine is essentially a computer being simulated on a computer - this is a way to be able to give someone access to computing resources with an extra level of flexibility, while isolating them from the original hardware. A simple virtual machine might be an independent environment running on a local desktop computer, or it could be on a large scale as in cloud computing, where a remote user can pay to use some amount of computing resources on a large system.

8. Protecting data and system resources is critical in digital systems.

Modern computing systems provide access to data and resources that if used inappropriately could breach privacy, provide unauthorised access to financial data or other resources, or even bring about physical harm. Security professionals often say that the weakest link in the security of a computer system is the user, and so it is essential that all computer users understand basic computer security principles. These principles include confidentiality (not allowing unauthorised access to information), availability (legitimate users can access their information), and integrity (the information is accurate).

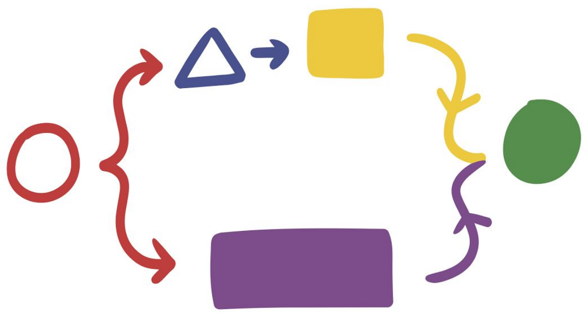
Computer users must be aware of and understand the tools and techniques that they can use to make their computing environment more secure. These tools include encryption methods, detecting and blocking attacks, authenticating who is accessing a system, and allowing users to recover from damage, whether malicious or accidental.

Digging deeper:

- The need for security is because of human issues. We work within codes of ethics, such as conferring some rights to privacy. Digital representations now control our ownership of property (particularly digital cash) and so traditional crime (such as theft) has now entered the digital world, and requires digital solutions. We also need to allow for human error, such as accidentally deleting or changing information.
- The importance of computer security highlights the value of educating people about it. For example, understanding the basic idea of Public Key Cryptography can help users to feel secure when communicating online, and understanding how passwords are stored can help a user to know *why* their password should have particularly characteristics.
- Cyber security is now a crucial part of a nation's defence. Nations can be attacked through the internet, and because so many aspects of our lives are controlled by digital systems, an attack on areas such as financial systems, power generation or food supply could have disastrous consequences.
- Another reason for attack is taking over computing resources to do work for the intruder e.g. loading programs on other people's computers to send spam or spread viruses.
- Balanced against this, we need to give appropriate people access at appropriate level as conveniently as possible.
- Security applies to communication systems as well as data storage; physical security (such as building access) is also commonly controlled digitally.
- Security doesn't just involve preventing intrusion; a "denial of service" attack simply overloads a system to prevent legitimate clients from using it, which could put a company out of business or prevent a government from doing its job.
- A key part of good security is having suitable backup, both for data and processing systems, and testing it to ensure that it would work if needed.
- Encryption is a central tool in computer security because it enables data to be changed so that an eavesdropper cannot easily read it. But this raises a new problem: how can people in different locations (such as an online store and a customer) set up a secure connection using secret encryption keys if those keys have to go through the same secure network that the eavesdropper is watching!? Public key cryptography provides a solution to this problem, where, remarkably, it is possible set up a communication and exchange messages in full view of an untrusted third party, without them being able to decode any of the messages.
- There are many cryptographic protocols that enable us to do things that might not seem possible, such as digital signatures that verify the source of a document; making cash payments without knowing who it came from; secure time-stamping where the time of an event or agreement can be recorded in a way that can't be modified; and electronic voting, where each person can verify that their vote was counted correctly, yet not be able to find out how others voted.

- Trying to find holes in security is an important role. A "white hat hacker" is someone who attempts to find security problems, but instead of exploiting the problem, they arrange to have it fixed before a "black hat hacker" can use it for negative purposes.

9. Time dependent operations in digital systems must be coordinated.



Digital systems have many components that can run independently; these components can be working in parallel, and on independent schedules. Parallelism occurs at many levels in digital systems, from instruction execution on a CPU, to multi-core systems in a laptop, to data being transmitted over a network through multiple routes, to large "big data" systems that process vast quantities of data in small chunks and combine the results.

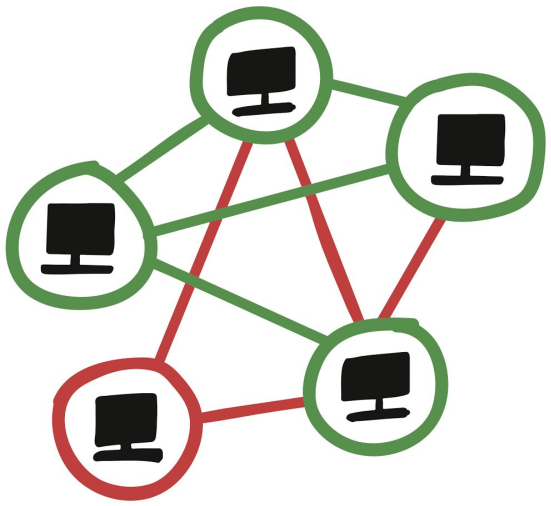
When a computational task is being spread over several independent parts of the system, considerable care is needed to make the most of the ability to spread the work over multiple devices. Problems need to be broken up into as many parts as possible that can be processed independently and recombined, and the dependency between these operations can restrict how easily a problem can be broken into parts.

Digging deeper:

- Computing power is commonly expanded by using multi-processor systems working in parallel; for example, even common desktop computers use multiple "cores" to spread the processing load across several processors. High performance computers can have hundreds or thousands of processors working in parallel, and "cloud" computing can involve arbitrary numbers of devices being applied to a task. A computer system might also involve multiple programs running independently, but communicating with each other (
- All of these situations require very careful coordination of the independent processes so that information is collated in the correct order when each process finishes. There is also the challenge of breaking a problem into many parts so that each part can be processed efficiently and independently, and then combined to provide a suitable solution to the problem.
- A key challenge is that applying twice as many processors doesn't necessarily solve a problem twice as fast; often there are parts of a solution that are needed before another part can be started on.
- Concurrent processes need to be designed carefully to avoid race conditions (where the order that events happen is critical), deadlocks (where two processes are both waiting for each other) and resource starvation (where some tasks prevent others from getting access to resources, such as a denial-of-service attack).

- Time dependent operations distinguish computing from algebra; for example, the sequence x=2; x=3 is meaningful in a computer program (x ends up with the value 3), but a contradiction in maths (x can't be 2 and 3 at the same time); and y=x+1; x=3 has an algebraic solution that is different to what would happen in a typical program because it depends on the order in which the instructions are executed.
- A related issue is "time slicing", where multiple processes are given turns at a small share of time on a machine, which makes it appear that each of them is running independently. This can happen on a single-user machine (e.g. running the clock at the same time as a word processor program), but also happens on computing systems that deal with multiple users (e.g. a website serving information to multiple clients). Each user may see the system as serving them alone, yet the server is designed to carefully switch between doing tasks for each user to give this appearance.

10. Digital systems communicate with each other using protocols.



Very few digital devices are an "island" - most are connected by wired or wireless networks, including local systems such as Bluetooth or USB connections. The goal is to get data through the networks as quickly as possible. However, networks are prone to errors from faulty components or transmission interference, and are also vulnerable to attack from people wanting to eavesdrop on the data or prevent it getting through.

Techniques are available to minimise these issues, to the extent that people use wireless data and the Internet for sending important and private information without being concerned about reliability and security. Protocols that ensure that the data has arrived safely and efficiently are essential for almost any situation: personal communications, commercial transactions, or military control all need to be sure that the data gets through reliably.

Digging deeper

- Communication protocols typically break data into packets, and use ideas like acknowledgement, time-outs, cryptographic protocols and compression to make sure data has got through correctly, efficiently and securely.
- These protocols are often expressed in "layers", from the details of the physical connectors and electronic specifications, through to details such as breaking data into packets, numbering them, and acknowledging that they have been received, and at the

highest level, specifying how a computer program can interface with a program on another computer.

- Protocols are the basis of a large variety of services such as the internet (e.g. the Internet Protocol, IP), the web (e.g. the Hypertext Transfer Protocol, http), secure data transfer (e.g. https) and e-mail (e.g the Internet Message Access Protocol, IMAP). Protocols are needed even to simply download a photo from a camera to a desktop computer (e.g. Bluetooth or USB).
- Protocols are created by humans, and are subject to negotiation; standards are created by considering competing options, and selecting a common approach that enables different devices to work together effectively. Sometimes the selection is done by committees, industry organisations and joint working groups (such as IPv6), and other times de facto standards emerge because of their wide adoption (such as HTML).