Algorithms

- Time to refocus on what we identified as the primary object of study in computer science: the algorithm
- We never really stopped talking about algorithms; at this point you can probably list a few:
    - Algorithms for interpreting bit sequences
    - The fetch-decode-execute algorithm
    - Algorithms for managing processes
    - Algorithms for network communication

Formal Definition

- We now move from our current *intuitive* understanding of an algorithm to a *formal* one
- We start with a formal definition:

  An *algorithm* is an <u>ordered</u> set of <u>unambiguous</u>, <u>executable</u> steps that defines a <u>terminating</u> process

- The underlined words constitute the core ideas in this definition, and separate the applied, informal uses of the term from its use in stricter, more technical, and scientific contexts

“What Something Is” vs. “How It Is Written”

- The fundamental distinction of *what* an object is vs. how that object is *represented* is crucial to computer science, especially for algorithms
- Ultimately, an algorithm is *abstract*: it exists in its own right (and in our minds) as a concept
- Thus, when we want to share an algorithm with someone (or some*thing*) else, it is crucial that we choose a representation that communicates the algorithm *completely* and *accurately* — easier said than done, as you might have noticed

Algorithms, Programs, and Processes

- Some algorithm representations can only be understood by humans: natural language (e.g., plain English), sketches, diagrams
- A *program* is an algorithm representation that is meant to be *readable* by a machine — which, today, typically refers to the computer as we know it
- An instance of a computer’s *performing* or *executing* that program is said to be a *process* — note that this is precisely how the term is used in the context of operating systems

Algorithm Representation

- We typically associate the act of *representing* concepts with a *language* — some set of symbols (e.g., "A B C D 1 2 3 ; : . ! ?," etc.) plus rules for putting them together (e.g., "a sequence of letters without spaces makes a word;" "a sequence of words ending with a period makes a sentence;" "to represent the idea of a *cat* in English, we put together the symbols *c-a-t*")
- Algorithm representation is no different, though as an object of formal study, some terms gain a precise meaning than you might have seen previously

- The fundamental symbols that we use for representing algorithms — analogous to our alphabet, numbers, and punctuation, and thus forming the building blocks of our representations — are called *primitives*
- These primitives are expected to be assembled in accordance with certain rules, analogous to our natural language grammar; this set of rules is called *syntax*
- When we put together these primitives and follow these rules, our intent is to convey something meaningful, such as the algorithm itself, or some of its key steps and concepts — this is called *semantics*
- The total system may be called a *programming language*
  - ◇ Informally, the syntax of a programming language determines how a program in that language *looks*, while the language's semantics determines what that program *means*

Representation Examples

- Some algorithm representations are, for now, understandable by humans only
    - Natural language
    - Diagrams (e.g., flowcharts), pictorial representations
- *Pseudocode* representation adds more formalism in both notation and in meaning, but remains primarily for human use
- When we finally represent the algorithm as a program in a particular *programming language*, we have a form that can be used by the computer

Key Elements of Any Algorithm

Pseudocode notation crystallizes certain concepts that are shared by almost all programming languages:

- *Values and expressions* — Notation for calculating and manipulating information, ranging from simple values (numbers, letters, text) to more complex structures (lists, data with distinct properties [e.g., an address has a number, street, city, and zip], and arbitrary combinations of these items [e.g., an address book of contacts, each with a list of addresses])
- *Naming and storage* — Notation for "holding on" to certain values and referring to them in various parts of an algorithm (e.g., variables like *x*, *index*, or *valueList*)
- *Sequencing* — Clear indication of the order in which an algorithm's steps will occur
- *Conditions and branching* — Different paths that an algorithm may follow based on some decision or state
- *Loops* — Repetition of a particular activity or set of activities, either depending on some condition or for a predetermined number of times (e.g., entering a password up to three times only; performing something once for every item in a list)

Algorithm Discovery

- All of this talk about *representing* an algorithm assumes that we even *have* an algorithm — and it's true that for many endeavors, algorithms *are* known
    - Converting temperature
    - Calculating an average or mean
    - Making change
    - Playing rock-scissors-paper…and many more
- Note how the struggle so far has been all about representation — in particular, how do we express these algorithms so that a machine can perform them

…but that's just a fraction of the fun!

- Representing an algorithm for a machine is called *programming*, and this is the activity that most people associate with computer science
- But think about it — if we don't know *what* the algorithm is in the first place, then there wouldn't *be* anything to program at all!
- In the end, the core innovations in computer science come from the *formulation* of an algorithm — independently of any programming language
    - How to manage multiple processes in an operating system
    - How to enable billions of computers to communicate
    - How to identify spam
- To paraphrase the Bard: "The *algorithm's* the thing"

Approaches to Algorithm Discovery

- In the end, there is no single guaranteed recipe for discovering an algorithm (think about that for a moment…that is like saying there is no known *algorithm* for discovering another algorithm!)
- But there are some general *strategies* that tend to increase our chances of finding such an algorithm
- Polya's approach specifies four phases: understand the problem, form plan(s) to solve the problem, execute the plan(s), then evaluate their effectiveness
    - Einstein was once asked what he would do if he was given five minutes to solve a problem; he said he would spend the first four minutes just *reading* the problem

- Two complementary philosophies have emerged for tackling computing problems:
    - *Top-down* — Break a problem up into smaller subproblems, then solve those first
    - *Bottom-up* — Look at specific instances of a problem, then see if those individualized solutions can lead you to a general answer

  Successful algorithm discovery usually takes a little bit of both approaches
- Analogous approaches can be found in techniques of mathematical proof:
    - *Induction* — Start with the simplest possible case, then build up a solution incrementally
    - *Contradiction* — See if you can solve an "opposite" problem, then "invert" the solution
- In the end, algorithm discovery remains very much an art, as evidenced by the wealth of algorithms that are yet to be discovered