Software: how we tell the machine what to do

- **hardware is a general purpose machine**
  - capable of doing instructions repetitively and very fast
  - doesn't do anything itself unless we tell it what to do

- **software: the instructions we want it to do**
  - different set of instructions
    -> different program
    -> machine behaves differently
  - program and data are stored in the same memory and manipulated by the same instructions

- **to tell a machine what to do,**
  - we have to spell out the steps in excruciating detail
  - programming languages help handle a lot of the details

Software roadmap

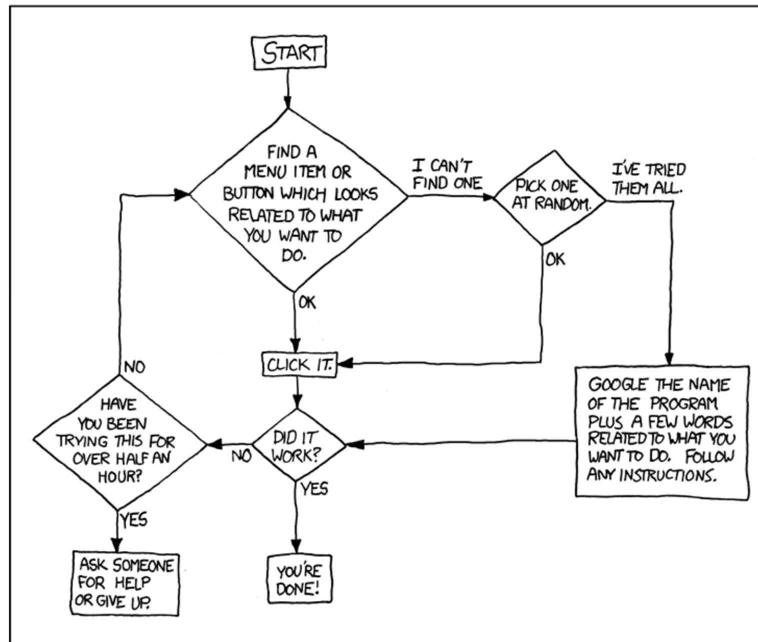
- **algorithm**
  - precise but abstract description of how to do some task
- **program**
  - precise concrete description of how to do some task on a real computer
- **programming languages**
  - precise notations for describing how to do tasks on a computer
    - e.g., Toy, Javascript
- **real programs (big software)**
  - operating systems
    - **file systems, databases**
  - applications

- **social / political / economic / legal issues**
  - interfaces
  - open source software
  - intellectual property, patents, copyrights

xkcd.com/627

DEAR VARIOUS PARENTS, GRANDPARENTS, CO-WORKERS, AND OTHER "NOT COMPUTER PEOPLE."

WE DON'T MAGICALLY KNOW HOW TO DO EVERYTHING IN EVERY PROGRAM. WHEN WE HELP YOU, WE'RE USUALLY JUST DOING THIS:



PLEASE PRINT THIS FLOWCHART OUT AND TAPE IT NEAR YOUR SCREEN. CONGRATULATIONS; YOU'RE NOW THE LOCAL COMPUTER EXPERT!

Algorithms

- **an <u>algorithm</u> is the computer science version of a really careful, precise, unambiguous recipe or procedure**
- **a sequence of steps that performs some computation**
- **each step is expressed in terms of basic operations whose meaning is completely specified**
  - basic operations or "primitive operations" are given
    - e.g., arithmetic operations
- **all possible situations are covered**
  - the algorithm never gets to a situation where it doesn't know what to do next
- **guaranteed to stop**
  - does not run forever

Some sample algorithms

- **compute average of two numbers**
  average = (first number + second number) / 2

- **compute average of N numbers**
  sum = 0
  for each number (from 1 to N)
    add i^{th} number to sum
  average = sum / N

- **convert decimal to binary**
  divide number by 2, write down remainder
  repeat until quotient becomes zero

- **many algorithms have this form:**
  - set up initial conditions (get started, get data to work on, ...)
  - repeat until some criterion is satisfied
  - finish the job

Linear time algorithms

- **lots of algorithms have this same basic form:**

  look at each item in turn
  do the same simple computation on each item:
    does it match something (looking up a name in a list of names)
    count it (how many items are in the list)
    count it if it meets some criterion (how many of some kind in the list)
    remember some property of items found (largest, smallest, …)
    transform it in some way (limit size, convert case of letters, …)

- **amount of work (running time) is proportional to amount of data**
  - twice as many items will take twice as long to process
  - computation time is linearly proportional to length of input

Log n algorithms

- **how do we find a name in a phone book?**
  - linear search requires looking at all the names
- **if the names are sorted into alphabetical order, we can use binary search, which is much faster than linear**
  - an example of a "divide and conquer" algorithm
- **data has to be sorted**
  - have to be able to access any data item equally quickly
  - "random access"
- **why is binary search faster than linear searching?**
  - each test / comparison cuts the number of things to search in half
- **how much faster is it?**
  - the number of comparison is approximately $\log_2 n$ for n items

Logarithms for COS 109

- **all logs in 109 are base 2**
- **all logs in 109 are integers**
- **if N is a power of 2 like 2^m, $\log_2$ of N is m**
- **if N is not a power of 2, $\log_2$ of N is**
  the number of bits needed to represent N
  the power of 2 that's bigger than N
  the number of times you can divide N by 2 before it becomes 0
- **you don't need a calculator for these!**
  - just figure out how many bits or what's the right power of 2
- **logs are related to exponentials: $\log_2 2^N$ is N**
- **it's the same as decimal, but with 2 instead of 10**

Algorithms for sorting

- **binary search needs sorted data**

- **how do we sort names into alphabetical order?**
- **how do we sort numbers into increasing or decreasing order?**
- **how do we sort a deck of cards?**

- **how many operations / comparisons does sorting take?**

- **"selection sort":**
  - find the smallest/earliest
    using a variant of "find the largest" algorithm
  - repeat on the remaining names
  - this is what bridge players typically do when organizing a hand

- **what other algorithms might work?**

How fast do these run?

- **searching an unordered/unsorted list of names**
  - time is proportional to length of the list
    because you might have to walk right to the end
  - twice as many items takes twice as long to search
- **searching a sorted list of names with binary search**
  - time is much faster (proportional to logarithm of length)
    because you can use divide-and-conquer to narrow the search
  - twice as many items needs only one more probe

- **sorting n items takes time proportional to n^2 with simple sorting algorithms like selection sort**
  - twice as many items takes 4 times as long to sort

- **there are much faster sorting algorithms (e.g., Quicksort)**
  - time proportional to n log n

Quicksort: an n log n sorting algorithm

- **make one pass through data, putting all small items in one pile and all large items on another pile**
  - there are now two piles, each with about 1/2 of the items
  - and each item in the first pile is smaller than any item in the second
- **make a second pass; for each pile, put all small items in one pile and all larger items in another pile**
  - there are now four piles, each with about 1/4 of the items
  - and each item in a pile is smaller than any item in later piles
- **repeat until there are n piles**
  - each item is now smaller than any item in a later pile

- **each pass looks at n items**
- **each pass divides each pile in half, stops when size is 1**
  - number of divisions is log n
- **n log n operations**

Complexity hierarchy (or part of it)

• log n	**logarithmic**	
• n	**linear**	**polynomial**
• n log n		**..**
• n^2	**quadratic**	**..**
• n^3	**cubic**	**..**
• 2^n	**exponential**	**(not polynomial)**

Algorithms in Computer Science

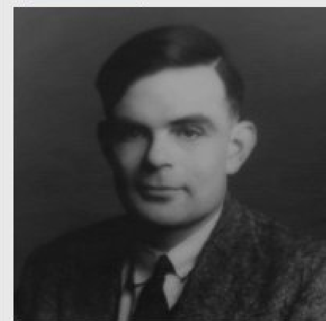
- **study and analysis of algorithms is a major component of CS courses**
  - what can be done (and what can't)
  - how to do it efficiently (fast, compact memory)
  - finding fundamentally new and better ways to do things
  - basic algorithms like searching and sorting
  - plus lots of applications with specific needs

- **big programs are usually a lot of simple, straightforward parts, often intricate, occasionally clever, very rarely with a new basic algorithm, sometimes with a new algorithm for a specific task**

P vs NP Problem

Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

- The Millennium Problems
- Official Problem Description — Stephen Cook
- Lecture by Vijaya Ramachandran at University of Texas (video)
- Minesweeper



Algorithms versus Programs

- **An algorithm is the computer science version of a really careful, precise, unambiguous recipe**
  - defined operations (primitives) whose meaning is completely known
  - defined sequence of steps, with all possible situations covered
  - defined condition for stopping

  - an idealized recipe

- **A program is an algorithm converted into a form that a computer can process directly**
  - like the difference between a blueprint and a building
  - has to worry about practical issues like finite memory, limited speed, erroneous data, etc.

  - a guaranteed recipe for a cooking robot