You are right to demand that an artist engage his work consciously, but you confuse two different things: solving the problem and correctly posing the question.

— Anton Chekhov, in a letter to A. S. Suvorin (October 27, 1888)

The more we reduce ourselves to machines in the lower things,
the more force we shall set free to use in the higher.

— Anna C. Brackett, *The Technique of Rest* (1892)

Arithmetic had entered the picture, with its many legs, its many spines and heads, its pitiless eyes made of zeroes. Two and two made four, was its message. But what if you didn't have two and two? Then things wouldn't add up.

— Margaret Atwood, *The Blind Assassin* (2000)

0 Introduction

0.1 What is an algorithm?

An algorithm is an unambiguous sequence of simple, mechanically executable instructions. Note that the word 'computer' doesn't appear anywhere in this definition; algorithms don't necessarily have anything to do with computers! For example, here is an algorithm for singing that annoying song '99 Bottles of Beer on the Wall', for arbitrary values of 99:

```
BOTTLESOFBEER(n):
  For i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."
  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```

The word 'algorithm' does *not* derive, as classically-trained algorithmophobes might guess, from the Greek root *algos* ($\alpha\lambda\gamma o\varsigma$), meaning 'pain'.[1] Rather, it is a corruption of the name of the 9th century Persian mathematician Abu 'Abd Allâh Muḥammad ibn Mûsâ al-Khwârizmî, which literally translates as "Mohammad, father of Adbdulla, son of Moses, the Kwârizmian".[2] (Kwârizm is an ancient city located in what is now the Xorazm Province of Uzbekistan.) Al-Khwârizmî is perhaps best known as the writer of the treatise *Kitab al-jabr wa'l-Muqâbala*, from which the modern word *algebra* derives. The word algorithm is a corruption of the older word *algorism* (by false connection to the Greek *arithmos* ($\alpha\rho\iota\theta\mu o\varsigma$), meaning 'number', and the aforementioned $\alpha\lambda\gamma o\varsigma$), used to describe the modern decimal system for writing and manipulating numbers—in particular, the use of a small circle or *sifr* to represent a missing quantity—which al-Khwârizmî brought into Persia from India. Thanks to the efforts of the medieval Italian mathematician Leonardo of Pisa, better known as Fibonacci, algorism began to replace the abacus as the preferred system of commercial calculation[3] in Europe in the late 12th century, although it was several more centuries before cyphers became truly ubiquitous. (Counting boards were used by the English and Scottish royal exchequers well into the 1600s.) So the word *algorithm* used to refer exclusively to mechanical pencil-and-paper methods for numerical calculations. People trained in the reliable execution of these methods were called—you guessed it—*computers*.

[1] Really. An *analgesic* is a medicine to remove pain; irrational fear of pain is called *algophobia*.

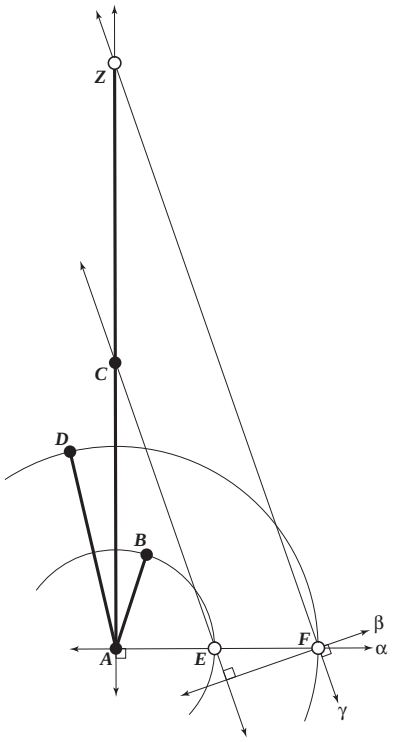
[2] Donald Knuth. Algorithms in modern mathematics and computer science. Chapter 4 in *Selected Papers on Computer Science*, Cambridge University Press, 1996. Originally published in 1981.

[3] from the Latin word *calculus*, meaning literally 'small rock', referring to the stones on a counting board, or abacus

Multiplication by compass and straightedge. However, algorithms have been with us since the dawn of civilization, centuries before Al-Khwârizmî and Fibonacci popularized the cypher. Here is an algorithm, popularized (but almost certainly not discovered) by Euclid about 2500 years ago, for multiplying or dividing numbers using a ruler and compass. The Greek geometers represented numbers using line segments of the appropriate length. In the pseudo-code below, CIRCLE(p, q) represents the circle centered at a point p and passing through another point q. Hopefully the other instructions are obvious.[4]

```
⟨⟨Construct the line perpendicular to ℓ and passing through P.⟩⟩
RIGHTANGLE(ℓ, P):
  Choose a point A ∈ ℓ
  A, B ← INTERSECT(CIRCLE(P, A), ℓ)
  C, D ← INTERSECT(CIRCLE(A, B), CIRCLE(B, A))
  return LINE(C, D)

⟨⟨Construct a point Z such that |AZ| = |AC||AD|/|AB|.⟩⟩
MULTIPLYORDIVIDE(A, B, C, D):
  α ← RIGHTANGLE(LINE(A, C), A)
  E ← INTERSECT(CIRCLE(A, B), α)
  F ← INTERSECT(CIRCLE(A, D), α)
  β ← RIGHTANGLE(LINE(E, C), F)
  γ ← RIGHTANGLE(β, F)
  return INTERSECT(γ, LINE(A, C))
```



Multiplying or dividing using a compass and straightedge.

This algorithm breaks down the difficult task of multiplication into simple primitive steps: drawing a line between two points, drawing a circle with a given center and boundary point, and so on. The primitive steps need not be quite this primitive, but each primitive step must be something that the person or machine executing the algorithm already knows how to do. Notice in this example that Euclid made constructing a right angle a primitive operation in the MULTIPLYORDIVIDE algorithm by (as modern programmers would put it) writing a subroutine.

Multiplication by duplation and mediation. Here is an even older algorithm for multiplying large numbers, sometimes called *(Russian) peasant multiplication*. A variant of this method was copied into the Rhind papyrus by the Egyptian scribe Ahmes around 1650 BC, from a document he claimed was (then) about 350 years old. This was the most common method of calculation by Europeans before Fibonacci's introduction of Arabic numerals. According to some sources, it was still being used in Russia well into the 20th century (along with the Julian calendar). This algorithm was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.[5]

[4] Euclid and his students almost certainly drew their constructions on an $\alpha\beta\alpha\xi$, a table covered in sand (or very small rocks). Over the next several centuries, the Greek *abax* evolved into the medieval European *abacus*.

[5] like the Apple II

```
PEASANTMULTIPLY(x, y):
  prod ← 0
  while x > 0
      if x is odd
          prod ← prod + y
      x ← ⌊x/2⌋
      y ← y + y
  return p
```

x	y	$prod$
		0
123	$+456$	$= 456$
61	$+912$	$= 1368$
30	~~1824~~	
15	$+3648$	$= 5016$
7	$+7296$	$= 12312$
3	$+14592$	$= 26904$
1	$+29184$	$= 56088$

The peasant multiplication algorithm breaks the difficult task of general multiplication into four simpler operations: (1) determining parity (even or odd), (2) addition, (3) duplation (doubling a number), and (4) mediation (halving a number, rounding down).[6] Of course a full specification of this algorithm requires describing how to perform those four 'primitive' operations. When executed by hand, peasant multiplication requires (a constant factor!) more paperwork, but the necessary operations are easier for humans to remember than the 10×10 multiplication table required by the American grade school algorithm.[7]

The correctness of peasant multiplication follows from the following recursive identity, which holds for any non-negative integers x and y:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

A bad example. Consider "Martin's algorithm":[8]

```
BECOMEAMILLIONAIREANDNEVERPAYTAXES:
  Get a million dollars.
  Don't pay taxes.
  If you get caught,
      Say "I forgot."
```

Pretty simple, except for that first step; it's a doozy. A group of billionaire CEOs would consider this an algorithm, since for them the first step is both unambiguous and trivial, but for the rest of us poor slobs who don't have a million dollars handy, Martin's procedure is too vague to be considered an algorithm. On the other hand, this is a perfect example of a *reduction*—it *reduces* the problem of being a millionaire and never paying taxes to the 'easier' problem of acquiring a million dollars. We'll see reductions over and over again in this class. As hundreds of businessmen and politicians have demonstrated, if you know how to solve the easier problem, a reduction tells you how to solve the harder one.

Martin's algorithm, like many of our previous examples, is not the kind of algorithm that computer scientists are used to thinking about, because it is phrased in terms of operations that are

[6] The ancient Egyptian version of this algorithm does not use mediation or parity, but it does use comparisons. To avoid halving, the algorithm pre-computes two tables by repeated doubling: one containing all the powers of 2 not exceeding x, the other containing the same powers of 2 multiplied by y. The powers of 2 that sum to x are found by subtraction, and the corresponding entries in the other table are added together to form the product. Egyptian scribes made large tables of powers of 2 to speed up these calculations.

[7] American school kids learn a variant of the *lattice* multiplication algorithm developed by Indian mathematicians and described by Fibonacci in *Liber Abaci*. The two algorithms are equivalent if the input numbers are represented in binary.

[8] S. Martin, "You Can Be A Millionaire", Saturday Night Live, January 21, 1978. Reprinted in *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

difficult for computers to perform. In this class, we'll focus (almost!) exclusively on algorithms that can be reasonably implemented on a computer. In other words, each step in the algorithm must be something that either is directly supported by common programming languages (such as arithmetic, assignments, loops, or recursion) or is something that you've already learned how to do in an earlier class (like sorting, binary search, or depth first search).

Congressional apportionment. Here is another good example of an algorithm that comes from outside the world of computing. Article I, Section 2 of the US Constitution requires that

> Representatives and direct Taxes shall be apportioned among the several States which may be included within this Union, according to their respective Numbers.... The Number of Representatives shall not exceed one for every thirty Thousand, but each State shall have at Least one Representative....

Since there are a limited number of seats available in the House of Representatives, exact proportional representation is impossible without either shared or fractional representatives, neither of which are legal. As a result, several different apportionment algorithms have been proposed and used to round the fractional solution fairly. The algorithm actually used today, called *the Huntington-Hill method* or *the method of equal proportions*, was first suggested by Census Bureau statistician Joseph Hill in 1911, refined by Harvard mathematician Edward Huntington in 1920, adopted into Federal law (2 U.S.C. §§2a and 2b) in 1941, and survived a Supreme Court challenge in 1992.[9] The input array $P[1..n]$ stores the populations of the n states, and R is the total number of representatives. Currently, $n = 50$ and $R = 435$.[10]

```
ApportionCongress(P[1..n], R):
  H ← NewMaxHeap
  for i ← 1 to n
      r[i] ← 1
      Insert(H, i, P[i]/√2)
  R ← R − n

  while R > 0
      s ← ExtractMax(H)
      r[s] ← r[s] + 1
      Insert(H, i, P[i]/√(r[i](r[i] + 1)))
      R ← R − 1

  return r[1..n]
```

Note that this description assumes that you know how to implement a max-heap and its basic operations NewMaxHeap, Insert, and ExtractMax. Moreover, the correctness of the algorithm doesn't depend at all on how these operations are implemented. The Census Bureau implements the max-heap as an unsorted array inside an Excel spreadsheet; you should have learned a more efficient solution in your undergraduate data structures class.

[9] Overruling an earlier ruling by a federal district court, the Supreme Court unanimously held that *any* apportionment method adopted in good faith by Congress is constitutional (*United States Department of Commerce v. Montana*). The congressional apportionment algorithm is described in detail at the U.S. Census Department web site http://www.census.gov/population/www/censusdata/apportionment/computing.html. A good history of the apportionment problem can be found at http://www.thirty-thousand.org/pages/Apportionment.htm. A report by the Congressional Research Service describing various apportionment methods is available at http://www.rules.house.gov/archives/RL31074.pdf.

[10] The DC Fair and Equal House Voting Rights Act of 2006, if it becomes law, would permanently increase the number of representatives to 437, exactly one of which would be allocated to to the District of Columbia. Thus, starting in 2010, the apportionment algorithm would be run with $n = 50$ and $R = 436$.

Combinatorial versus numerical. This class will focus specifically on *combinatorial* algorithms, as opposed to *numerical* algorithms. The distinction is fairly artificial, but essentially, numerical algorithms are used to approximate computation with ideal real numbers on finite precision computers. For example, here's a numerical algorithm to compute the square root of a number to a given precision. This algorithm works remarkably quickly—every iteration doubles the number of correct digits.

```
SQUAREROOT(x, ε):
  s ← 1
  while |s − x/s| > ε
      s ← (s + x/s)/2
  return s
```

The output of a numerical algorithm is necessarily an approximation to some ideal mathematical object. Any number that's close enough to the ideal answer is a correct answer. Combinatorial algorithms, on the other hand, manipulate discrete objects like arrays, lists, trees, and graphs that can be represented *exactly* on a digital computer.

0.2 Writing down algorithms

Algorithms are *not* programs; they should not be described in a particular programming language. The whole *point* of this course is to develop computational techniques that can be used in *any* programming language.[11] The idiosyncratic syntactic details of C, Java, Python, Scheme, Visual Basic, ML, Smalltalk, Javascript, Forth, TEX, COBOL, Intercal, or Brainfuck[12] are of absolutely no importance in algorithm design, and focusing on them will only distract you from what's really going on.[13] What we really want is closer to what you'd write in the *comments* of a real program than the code itself.

On the other hand, a plain English prose description is usually not a good idea either. Algorithms have a lot of structure—especially conditionals, loops, and recursion—that are far too easily hidden by unstructured prose. Like any language spoken by humans, English is full of ambiguities, subtleties, and shades of meaning, but algorithms must be described as accurately as possible. Finally and more seriously, many people have a tendency to describe loops informally: "Do this first, then do this second, and so on." As anyone who has taken one of those 'what comes next in this sequence?' tests already knows, specifying what happens in the first couple of iterations of a loop doesn't say much about what happens later on.[14] Phrases like 'and so on' or 'do X over and over'

[11] See http://www.ionet.net/~timtroyr/funhouse/beer.html for implementations of the BOTTLESOFBEER algorithm in over 200 different programming languages.

[12] Brainfuck is the well-deserved name of a programming language invented by Urban Mueller in 1993. Brainfuck programs are written entirely using the punctuation characters `<>+-,.[]`, each representing a different operation (roughly: shift left, shift right, increment, decrement, input, output, begin loop, end loop). See http://www.catseye.mb.ca/esoteric/bf/ for a complete definition, sample programs, an interpreter (written in just 230 characters of C), and related shit.

[13] This is, of course, a matter of religious conviction. Linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states that people think only in the categories imposed by their languages. According to this hypothesis, some concepts in one language simply cannot be understood by speakers of other languages, not just because of technological advancement—How would you translate "jump the shark" or "blog" into ancient Greek?—but because of inherent structural differences between languages and cultures. Anne Rice espoused a similar idea in her later Lestat books. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is some strength to this idea when applied to programming languages. (What's the Y combinator, again? How do templates work?) Fortunately, those differences are generally too subtle to have much impact in *this* class.

[14] See http://www.research.att.com/~njas/sequences/.

or 'et cetera' are a good indication that the algorithm *should* have been described in terms of loops or recursion, and the description should have specified what happens in a *generic* iteration of the loop. Similarly, the appearance of the phrase 'and so on' in a proof is a good indication that the proof *should* have been done by induction!

The best way to write down an algorithm is using pseudocode. Pseudocode uses the structure of formal programming languages and mathematics to break the algorithm into one-sentence steps, but those sentences can be written using mathematics, pure English, or some mixture of the two. Exactly how to structure the pseudocode is a personal choice, but the overriding goal should be clarity and precision. Here are the basic rules I follow:

- Use standard imperative programming keywords (if/then/else, while, for, repeat/until, case, return) and notation (variable←value, array[index], pointer→field, function(args), etc.)
- The block structure should be visible from across the room. Indent everything carefully and consistently. Don't use syntactic sugar (like C/C++/Java braces or Pascal/Algol begin/end tags) unless the pseudocode is absolutely unreadable without it.
- *Don't* typeset keywords in a different **font** or `style`. Changing type style emphasizes the keywords, making the reader think the syntactic sugar is actually important—it isn't!
- Each statement should fit on one line, and each line should contain only one statement. (The only exception is extremely short and similar statements like $i \leftarrow i+1;\ j \leftarrow j-1;\ k \leftarrow 0$.)
- Put each structuring statement (for, while, if) on its own line. The order of nested loops matters a great deal; make it absolutely obvious.
- Use short but mnemonic algorithm and variable names. Absolutely *never* use pronouns!

A good description of an algorithm reveals the internal structure, hides irrelevant details, and can be implemented easily by any competent programmer in any programming language, even if they don't understand why the algorithm works. Good pseudocode, like good code, makes the algorithm much easier to understand and analyze; it also makes mistakes much easier to spot. The algorithm descriptions in the textbooks and lecture notes are good examples of what we want to see on your homeworks and exams.

0.3 Analyzing algorithms

It's not enough just to write down an algorithm and say 'Behold!' We also need to convince ourselves (and our graders) that the algorithm does what it's supposed to do, and that it does it quickly.

Correctness: In the real world, it is often acceptable for programs to behave correctly most of the time, on all 'reasonable' inputs. Not in this class; our standards are higher[15]. We need to *prove* that our algorithms are correct on *all possible* inputs. Sometimes this is fairly obvious, especially for algorithms you've seen in earlier courses. But many of the algorithms we will discuss in this course will require some extra work to prove. Correctness proofs almost always involve induction. We *like* induction. Induction is our *friend*.[16]

Before we can formally prove that our algorithm does what we want it to, we have to formally state what we want the algorithm to do! Usually problems are given to us in real-world terms,

[15] or at least different

[16] If induction is *not* your friend, you will have a hard time in this course.

not with formal mathematical descriptions. It's up to us, the algorithm designer, to restate the problem in terms of mathematical objects that we can prove things about: numbers, arrays, lists, graphs, trees, and so on. We also need to determine if the problem statement makes any hidden assumptions, and state those assumptions explicitly. (For example, in the song "n Bottles of Beer on the Wall", n is always a positive integer.) Restating the problem formally is not only required for proofs; it is also one of the best ways to really understand what the problems is asking for. The hardest part of solving a problem is figuring out the right way to ask the question!

An important distinction to keep in mind is the distinction between a problem and an algorithm. A problem is a task to perform, like "Compute the square root of x" or "Sort these n numbers" or "Keep n algorithms students awake for t minutes". An algorithm is a set of instructions that you follow if you want to execute this task. The same problem may have hundreds of different algorithms.

Running time: The usual way of distinguishing between different algorithms for the same problem is by how fast they run. Ideally, we want the fastest possible algorithm for our problem. In the real world, it is often acceptable for programs to run efficiently most of the time, on all 'reasonable' inputs. Not in this class; our standards are different. We require algorithms that *always* run efficiently, even in the worst case.

But how do we measure running time? As a specific example, how long does it take to sing the song BOTTLESOFBEER(n)? This is obviously a function of the input value n, but it also depends on how quickly you can sing. Some singers might take ten seconds to sing a verse; others might take twenty. Technology widens the possibilities even further. Dictating the song over a telegraph using Morse code might take a full minute per verse. Ripping an mp3 over the Web might take a tenth of a second per verse. Duplicating the mp3 in a computer's main memory might take only a few microseconds per verse.

What's important here is how the singing time changes as n grows. Singing BOTTLESOFBEER($2n$) takes about twice as long as singing BOTTLESOFBEER(n), no matter what technology is being used. This is reflected in the asymptotic singing time $\Theta(n)$. We can measure time by counting how many times the algorithm executes a certain instruction or reaches a certain milestone in the 'code'. For example, we might notice that the word 'beer' is sung three times in every verse of BOTTLESOFBEER, so the number of times you sing 'beer' is a good indication of the total singing time. For this question, we can give an exact answer: BOTTLESOFBEER(n) uses exactly $3n+3$ beers.

There are plenty of other songs that have non-trivial singing time. This one is probably familiar to most English-speakers:

```
NDAYSOFCHRISTMAS(gifts[2..n]):
  for i ← 1 to n
      Sing "On the ith day of Christmas, my true love gave to me"
      for j ← i down to 2
          Sing "j gifts[j]"
      if i > 1
          Sing "and"
      Sing "a partridge in a pear tree."
```

The input to NDAYSOFCHRISTMAS is a list of $n-1$ gifts. It's quite easy to show that the singing time is $\Theta(n^2)$; in particular, the singer mentions the name of a gift $\sum_{i=1}^{n} i = n(n+1)/2$ times (counting the partridge in the pear tree). It's also easy to see that during the first n days of Christmas, my true love gave to me exactly $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6 = \Theta(n^3)$ gifts. Other songs that take quadratic time to sing are "Old MacDonald", "There Was an Old Lady Who Swallowed

a Fly", "Green Grow the Rushes O", "The Barley Mow", "Echad Mi Yode'a" ("Who knows one?"), "Allouette", "Ist das nicht ein Schnitzelbank?"[17] etc. For details, consult your nearest preschooler.

```
OLDMACDONALD(animals[1..n], noise[1..n]):
  for i ← 1 to n
      Sing "Old MacDonald had a farm, E I E I O"
      Sing "And on this farm he had some animals[i], E I E I O"
      Sing "With a noise[i] noise[i] here, and a noise[i] noise[i] there"
      Sing "Here a noise[i], there a noise[i], everywhere a noise[i] noise[i]"
      for j ← i − 1 down to 1
          Sing "noise[j] noise[j] here, noise[j] noise[j] there"
          Sing "Here a noise[j], there a noise[j], everywhere a noise[j] noise[j]"
      Sing "Old MacDonald had a farm, E I E I O."
```

```
ALLOUETTE(lapart[1..n]):
  Chantez „Allouette, gentille allouette, allouette, je te plumerais."
  pour tout i de 1 á n
      Chantez „Je te plumerais lapart[i], je te plumerais lapart[i]."
      pour tout j de i − 1 á bas á 1
          Chantez „Et lapart[j], et lapart[j],"
      Chantez „Oooooooo!"
      Chantez „Allouette, gentille alluette, allouette, je te plumerais."
```

For a slightly more complicated example, consider the algorithm APPORTIONCONGRESS. Here the running time obviously depends on the implementation of the max-heap operations, but we can certainly bound the running time as $O(N+RI+(R-n)E)$, where N is the time for a NEWMAXHEAP, I is the time for an INSERT, and E is the time for an EXTRACTMAX. Under the reasonable assumption that $R > 2n$ (on average, each state gets at least two representatives), this simplifies to $O(N+R(I+E))$. The Census Bureau uses an unsorted array of size n, for which $N = I = \Theta(1)$ (since we know a priori how big the array is), and $E = \Theta(n)$, so the overall running time is $\Theta(Rn)$. This is fine for the federal government, but if we want to be more efficient, we can implement the heap as a perfectly balanced n-node binary tree (or a heap-ordered array). In this case, we have $N = \Theta(1)$ and $I = R = O(\log n)$, so the overall running time is $\Theta(R \log n)$.

Incidentally, there is a faster algorithm for apportioning Congress. I'll give extra credit to the first student who can find the faster algorithm, analyze its running time, and prove that it always gives exactly the same results as APPORTIONCONGRESS.

Sometimes we are also interested in other computational resources: space, randomness, page faults, inter-process messages, and so forth. We use the same techniques to analyze those resources as we use for running time.

0.4 Why are we here, anyway?

This class is ultimately about learning two skills that are crucial for computer scientists: how to *think* about algorithms and how to *talk* about algorithms. Along the way, you'll pick up a bunch of algorithmic facts—mergesort runs in $\Theta(n \log n)$ time; the amortized time to search in a splay tree is $O(\log n)$; greedy algorithms usually don't produce optimal solutions; the traveling salesman problem is NP-hard—but these aren't the point of the course. You can always look up mere facts in a textbook or on the web, provided you have enough intuition and experience to know what to look

[17] Wakko: Ist das nicht Otto von Schnitzelpusskrankengescheitmeyer?
Yakko and Dot: Ja, das ist Otto von Schnitzelpusskrankengescheitmeyer!!

for. That's why we let you bring cheat sheets to the exams; we don't want you wasting your study time trying to memorize all the facts you've seen. You'll also practice a lot of algorithm design and analysis skills—finding useful (counter)examples, developing induction proofs, solving recurrences, using big-Oh notation, using probability, giving problems crisp mathematical descriptions, and so on. These skills are *very* useful, but they aren't really the point of the course either. At this point in your educational career, you should be able to pick up those skills on your own, once you know what you're trying to do.

The first main goal of this course is to help you develop algorithmic *intuition*. How do various algorithms really work? When you see a problem for the first time, how should you attack it? How do you tell which techniques will work at all, and which ones will work best? How do you judge whether one algorithm is better than another? How do you tell whether you have the best possible solution?

Our second main goal is to help you develop algorithmic *language*. It's not enough just to understand how to solve a problem; you also have to be able to explain your solution to somebody else. I don't mean just how to turn your algorithms into working code—despite what many students (and inexperienced programmers) think, 'somebody else' is *not* just a computer. Nobody programs alone. Code is read far more often than it is written, or even compiled. Perhaps more importantly in the short term, explaining something to somebody else is one of the best ways of clarifying your own understanding. As Richard Feynman apocryphally put it, "If you can't explain what you're doing to your grandmother, you don't understand it."

Unfortunately, there is no systematic procedure—no algorithm—to determine which algorithmic techniques are most effective at solving a given problem, or finding good ways to explain, analyze, optimize, or implement a given algorithm. Like many other human activities (music, writing, juggling, acting, martial arts, sports, cooking, programming, teaching, etc.), experts will disagree on the relative values of different techniques. Ultimately, the *only* way to master these skills is to make them your own, through practice, practice, and more practice. We *can't* teach you how to do well in this class. All we can do is lay out a few tools, show you how to use them, create opportunities for you to practice, and give you feedback based on our own experience and intuition. The rest is up to you.

Good algorithms are extremely useful, elegant, surprising, deep, even beautiful. But most importantly, algorithms are *fun*!! I hope this course will inspire at least some you to come play!

© Copyright 2006 Jeff Erickson. Released under a Creative Commons Attribution-NonCommercial-ShareAlike 2.5 License (http://creativecommons.org/licenses/by-nc-sa/2.5/)