Partly because of his computational skills, Gerbert, in his later years, was made Pope by Otto the Great, Holy Roman Emperor, and took the name Sylvester II. By this time, his gift in the art of calculating contributed to the belief, commonly held throughout Europe, that he had sold his soul to the devil.

— Dominic Olivastro, *Ancient Puzzles*, 1993

0 Introduction (August 29)

0.1 What is an algorithm?

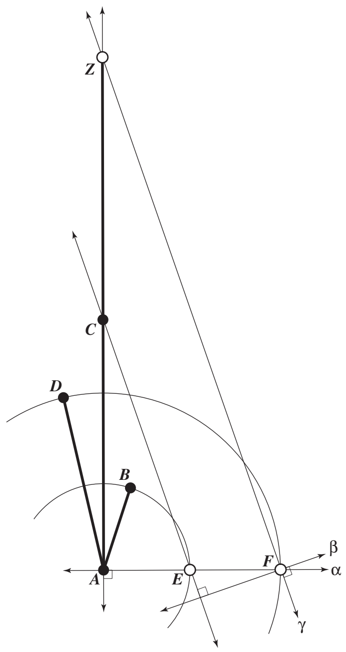
This is a course about algorithms, specifically combinatorial algorithms. An algorithm is a set of simple, unambiguous, step-by-step instructions for accomplishing a specific task. Note that the word "computer" doesn't appear anywhere in this definition; algorithms don't necessarily have anything to 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, x bottles of beer on the wall."
```

Algorithms have been with us since the dawn of civilization. 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 right length. In the pseudo-code below, $\textsc{Circle}(p, q)$ represents the circle centered at a point p and passing through another point q; hopefully the other instructions are obvious.

```
⟨⟨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 straight-edge.

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 we have made constructing a right angle a primitive operation in the MULTIPLYORDIVIDE algorithm by writing a subroutine.

As a bad example, consider "Martin's algorithm":[1]

```
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.]

Although most of the previous examples are algorithms, they're not the kind of algorithms that computer scientists are used to thinking about. 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 your favorite programming language (arithmetic, assignments, loops, recursion, etc.) or is something that you've already learned how to do in an earlier class (sorting, binary search, depth first search, etc.).

For example, here's the algorithm that's actually used to determine the number of congressional representatives assigned to each state.[2] 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 $R = 435$.)

```
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]
```

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

[2]The congressional apportionment algorithm is described in detail at http://www.census.gov/population/www/censusdata/apportionment/computing.html, and some earlier algorithms are described at http://www.census.gov/population/www/censusdata/apportionment/history.html.

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.[3] The Census Bureau implements the max-heap as an unsorted array, probably inside an Excel spreadsheet. (You should have learned a more efficient solution in CS 225.)

So what's a *combinatorial* algorithm? The distinction is fairly artificial, but basically, this means something distinct from a numerical algorithm. 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 never be specified in a particular programming language. The whole *point* of this course is to develop computational techniques that can be used in *any* programming language.[4] The idiosyncratic syntactic details of C, Java, Visual Basic, ML, Smalltalk, Intercal, or Brainfunk[5] are of absolutely no importance in algorithm design, and focusing on them will only distract you from what's really going on. 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.[6] Phrases like 'and so on' or 'do X over and over' or 'et cetera' are a good indication that the algorithm *should* have been described in terms of

[3] However, as several students pointed out, the algorithm's correctness does depend on the populations of the states being not too different. If Congress had only 384 representatives, then the 2000 Census data would have given Rhode Island only one representative! If state populations continue to change at their current rates, The United States have have a serious constitutional crisis right after the 2030 Census. I can't wait!

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

[5] Pardon my thinly bowdlerized Anglo-Saxon. Brainfork is the well-deserved name of a programming language invented by Urban Mueller in 1993. Brainflick 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 of Brainfooey, sample Brainfudge programs, a Brainfiretruck interpreter (written in just 230 characters of C), and related shit.

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

loops or recursion, and the description should have specified what happens in a *generic* iteration of the loop.[7]

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 here are the basic rules I follow:

- Use standard imperative programming keywords (if/then/else, while, for, repeat/until, case, return) and notation (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, 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[8]. 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*.[9]

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,

[7]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!

[8]or at least different

[9]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.

Nevertheless, 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" (which we'll see in Homework 1), "Echad Mi Yode'a" ("Who knows one?"), "Allouette", "Ist das nicht ein Schnitzelbank?"[10] etc. For details, consult your nearest pre-schooler.

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, disk swaps, concurrency, 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?

We will try to teach you two things in CS373: 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 $O(\log n)$; the traveling salesman problem is NP-hard—but these aren't the point of the course. You can always look up facts in a textbook, provided you have the intuition to know what to look 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, and so on. These skills are 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 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 if you have the best possible solution?

The flip side of this goal is developing 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 code—despite what many students (and inexperienced programmers) think, 'somebody else' is *not* just a computer. Nobody programs alone. Perhaps more importantly in the short term, explaining something to somebody else is one of the best ways of clarifying your own understanding.

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

You'll also get a chance to develop brand new algorithms and algorithmic techniques on your own. Unfortunately, this is not the sort of thing that we can really teach you. All we can really do is lay out the tools, encourage you to practice with them, and give you feedback.

Good algorithms are extremely useful, but they can also be elegant, surprising, deep, even beautiful. But most importantly, algorithms are *fun*!! Hopefully this class will inspire at least a few of you to come play!