We should explain, before proceeding, that it is not our object to consider this program with reference to the actual arrangement of the data on the Variables of the engine, but simply as an abstract question of the nature *and* number *of the* operations *required to be perfomed during its complete solution.*

— Ada Augusta Byron King, Countess of Lovelace, translator's notes for Luigi F. Menabrea, "Sketch of the Analytical Engine invented by Charles Babbage, Esq." (1843)

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)

0 Introduction

0.1 What is an algorithm?

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions. 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 algorithmophobic classicists might guess, from the Greek root *algos* ($\alpha\lambda\gamma o\varsigma$), meaning 'pain'. Rather, it is a corruption of the name of the 9th century Persian mathematician Abū 'Abd Allāh Muḥammad ibn Mūsā al-Khwārizmī.[1] Al-Khwārizmī is perhaps best known as the writer of the treatise *Al-Kitāb al-mukhtaṣar fīhīsāb al-ğabr wa'l-muqābala*[2], 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 perhaps 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-Khwārizmī 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.) Thus, until very recently, the word *algorithm* referred exclusively to pencil-and-paper methods for numerical calculations. People trained in the reliable execution of these methods were called—you guessed it—*computers*.

[1] 'Mohammad, father of Adbdulla, son of Moses, the Kwārizmian'. Kwārizm is an ancient city, now called Khiva, in the Khorezm Province of Uzbekistan.

[2] 'The Compendious Book on Calculation by Completion and Balancing'.

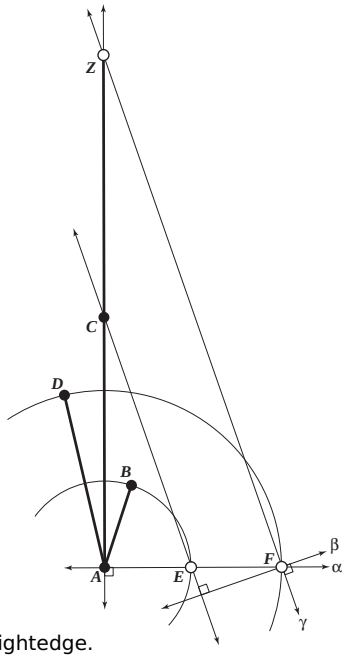
[3] from the Latin word *calculus*, meaning literally 'small rock', referring to the stones on a counting board, or abacus

0.2 A Few Simple Examples

Multiplication by compass and straightedge. Although they have only been an object of formal study for a few decades, algorithms have been with us since the dawn of civilization, for centuries before Al-Khwārizmī 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 a series of simple primitive operations: drawing a line between two points, drawing a circle with a given center and boundary point, and so on. These primitive steps are quite non-trivial to execute on a modern digital computer, but this algorithm wasn't designed for a digital computer; it was designed for the Platonic Ideal Classical Greek Mathematician, wielding the Platonic Ideal Compass and the Platonic Ideal Straightedge. In this example, Euclid first defines a new primitive operation, constructing a right angle, 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; it was still being used in Russia, along with the Julian calendar, well into the 20th century. This algorithm was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.

[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*.

```
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).[5] Of course a full specification of this algorithm requires describing how to perform those four 'primitive' operations. Peasant multiplication requires (a constant factor!) more paperwork to execute by hand, but the necessary operations are easier (for humans) to remember than the 10×10 multiplication table required by the American grade school algorithm.[6]

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":[7]

```
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, 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 difficult for computers to perform. In this class, we'll focus (almost!) exclusively on algorithms that can be

[5] The version of this algorithm actually used in ancient Egypt 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 then found by greedy subtraction, and the corresponding entries in the other table are added together to form the product.

[6] 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.

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

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.[8] 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]
```

Note that this description assumes that you know how to implement a max-heap and its basic operations NewMaxHeap, Insert, and ExtractMax. (The actual law doesn't make those assumptions, of course.) 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 stored in a column of an Excel spreadsheet; you should have learned a more efficient solution in your undergraduate data structures class.

[8] 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 current congressional apportionment algorithm is described in gruesome 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.

0.3 Writing down algorithms

Computer programs are concrete representations of algorithms, but 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.[9] The idiosyncratic syntactic details of C, Java, Python, Ruby, Erlang, OcaML, Scheme, Visual Basic, Smalltalk, Javascript, Forth, TEX, COBOL, Intercal, or Brainfuck[10] are of little or no importance in algorithm design, and focusing on them will only distract you from what's *really* going on.[11] 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 natural languags, 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 few iterations of a loop doesn't say much about what happens in later iterations.[12] Phrases like 'and so on' or 'repeat this for all n' are good indicators that the algorithm should have been described in terms of loops or recursion, and the description should have specified a *generic* iteration of the loop. Similarly, the appearance of the phrase 'and so on' in a proof almost always means 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 an appropriate 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 guidelines I follow:

- Use standard imperative programming keywords (if/then/else, while, for, repeat/until, case, return) and notation ($variable \leftarrow value$, $Array[index]$, $function(argument)$, $bigger > smaller$, etc.). Keywords should be standard English words: write 'else if' instead of 'elif'.
- Use standard mathematical notation for standard mathematical stuff. For example, write $\sqrt{x}$ and a^b and π instead of $sqrt(x)$ and $power(a, b)$ and pi. (One exception: *never* use $\forall$ to represent a for-loop!)
- Avoid mathematical notation if English is clearer. For example, 'Insert a into X' is often preferable to $\textsc{Insert}(X, a)$ or $X \leftarrow X \cup \{a\}$.

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

[10] Brainfuck is the well-deserved name of a programming language invented by Urban Müller 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://esolangs.org/wiki/Brainfuck for a complete language description; links to several interpreters, compilers, and sample programs; and lots of related shit.

[11] This is, of course, a matter of religious conviction. Linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states (more or less) that people think only in the categories imposed by their languages. According to an extreme formulation of this principle, 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 Aramaic?—but because of inherent structural differences between languages and cultures. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is admittedly some strength to this idea when applied to different programming paradigms. (What's the Y combinator, again? How do templates work? What's an Abstract Factory?) Fortunately, those differences are generally too subtle to have much impact in *this* class.

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

- Indent everything carefully and consistently; the block structure should be visible from across the room. This is especially important for nested loops and conditionals. *Don't* use unnecessary syntactic sugar (like braces or begin/end tags).
- *Don't* typeset keywords in a different **font** or `style`. Changing type style emphasizes the keywords, making the reader think the syntactic sugar is important. It isn't!
- Each statement should fit on one line, and each line should contain either exactly one statement or exactly one structuring element (for, while, if). (I sometimes make an exception for short and similar statements like $i \leftarrow i+1;\ j \leftarrow j-1;\ k \leftarrow 0$.)
- Use short, 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 and correctly by any competent programmer in *their* favorite 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 these lecture notes are (hopefully) good examples of what we want to see on your homeworks and exams.

0.4 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 efficiently.

Correctness: In some application settings, it is acceptable for programs to behave correctly most of the time, on all 'reasonable' inputs. Not in this class; we require algorithms that are correct for *all possible* inputs. Moreover, we must *prove* that they're correct; trusting our instincts, or trying a few test cases, isn't good enough.[13] Sometimes correctness is fairly obvious, especially for algorithms you've seen in earlier courses. On the other hand, 'obvious' is all too often a synonym for 'wrong'. 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*.[14]

But before we can formally prove that our algorithm does what we want it to do, we have to formally state what we want it to do! Usually problems are given to us in real-world terms, not with formal mathematical descriptions. It's up to us, the algorithm designers, to restate these problems 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 answering any question is figuring out the right way to ask it!

It is important to remember 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 to follow if you want to accomplish this task. The same problem may have hundreds of different algorithms; the same algorithm may solve hundreds of different problems.

[13] I say we take off and nuke the entire site from orbit. It's the only way to be sure.

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

Running time: The most common way of ranking different algorithms for the same problem is by how fast they run. Ideally, we want the fastest possible algorithm for our problem. In many application settings, it is acceptable for programs to run efficiently most of the time, on all 'reasonable' inputs. Not in this class; 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. Downloading 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 had a Farm", "There Was an Old Lady Who Swallowed a Fly", "The House that Jack Built", "Green Grow the Rushes O", "Eh, Compare!", "Alouette", "Echad Mi Yodea", "Chad Gadya", and "Ist das nicht ein Schnitzelbank?"[15] For further 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."
```

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

```
ALOUETTE(lapart[1..n]):
  Chantez « Alouette, gentille alouette, alouette, 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 « Alouette, gentille alluette, alouette, 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 $\Theta(R \log n)$.

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.

0.5 A Longer Example: Stable Marriage

Every year, thousands of new doctors must obtain internships at hospitals around the United States. During the first half of the 20th century, competition among hospitals for the best doctors led to earlier and earlier offers of internships, sometimes as early as the second year of medical school, along with tighter deadlines for acceptance. In the 1940s, medical schools agreed not to release information until a common date during their students' fourth year. In response, hospitals began demanding faster decisions. By 1950, hospitals would regularly call doctors, offer them an internship, and demand an *immediate* response. Interns were forced to gamble if their third-choice hospital called first—accept and risk losing a better opportunity later, or reject and risk having no position at all.

Finally, a central clearinghouse for internship assignments, now called the National Resident Matching Program, was established in the early 1950s. Each year, doctors submit a ranked list of all hospitals where they would accept an internship, and each hospital submits a ranked list of doctors they would accept as interns. The NRMP then computes an assignment of interns to hospitals that satisfies the following *stability* requirement. For simplicity, let's assume that there are n doctors and n hospitals; each hospital offers exactly one internship; each doctor ranks all hospitals and vice versa; and finally, there are no ties in the doctors' and hospitals' rankings.[16] We say that a matching of doctors to hospitals is ***unstable*** if there are two doctors α and β and two hospitals A and B, such that

- α is assigned to A, and β is assigned to B;
- α prefers B to A, and B prefers α to β.

In other words, α and B would both be happier with each other than with their current assignment. The goal of the Resident Match is a *stable* matching, in which no doctor or hospital has an incentive to cheat the system. At first glance, it is not clear that a stable matching exists!

[16] In reality, most hospitals offer multiple internships, Each doctor ranks only a subset of the hospitals and vice versa, and there are typically more internships than interested doctors. And then it starts getting complicated.

In 1952, the NRMP adopted the "Boston Pool" algorithm to assign interns, so named because it had been previously used by a regional clearinghouse in the Boston area. The algorithm is often inappropriately attributed to David Gale and Lloyd Shapley, who formally analyzed the algorithm and first proved that it computes a stable matching in 1962.[17] The algorithm proceeds in rounds until every position has been filled. In each round:

1. Some unassigned hospital offers its position to the best doctor (according to the hospital's preference list) who has not already rejected it.

2. Each doctor is always assigned to the best hospital (according to the doctor's preference list) that has made her an offer so far. Thus, whenever a doctor receives an offer that she likes more than her current assignment, her assignment changes.

For example, suppose there are three doctors α, β, γ, δ, and three hospitals A, B, C, D with the following preference lists:

α	β	γ	δ	A	B	C	D
A	A	B	D	δ	β	δ	γ
B	D	A	B	γ	δ	α	β
C	C	C	C	β	α	β	α
D	B	D	A	α	γ	γ	δ

The algorithm might proceed as follows:

1. A makes an offer to δ.
2. B makes an offer to β.
3. C makes an offer to δ, who rejects her earlier offer from A.
4. D makes an offer to γ. (From this point on, because there is only one unmatched hospital, the algorithm has no more choices.)
5. A makes an offer to γ, who rejects her earlier offer from D.
6. D makes an offer to β, who rejects her earlier offer from B.
7. B makes an offer to δ, who rejects her earlier offer from C.
8. C makes an offer to α.

At this point we have the assignment $(\alpha, C), (\beta, D), (\gamma, A), (\delta, B)$. You can verify by brute force that this matching is stable, despite the fact that no doctor was hired by her favorite hospital, and no hospital hired its favorite doctor.

Analyzing the algorithm is straightforward. Since each hospital makes an offer to each doctor at most once, the algorithm requires at most n^2 rounds. In an actual implementation, each doctor and hospital can be identified by a unique integer between 1 and n, and the preference lists can be represented as two arrays $DocPref[1..n][1..n]$ and $HosPref[1..n][1..n]$, where $DocPref[\alpha][r]$ represents the rth hospital

[17] Gale and Shapely used the metaphor of college admissions. The "Gale-Shapely algorithm" is a prime instance of Stigler's Law of Eponymy: No scientific discovery is named after its original discoverer. In his 1980 paper that gives the law its name, the statistician Stephen Stigler claimed that this law was first proposed by sociologist Robert K. Merton, although similar sentiments were previously attributed to Vladimir Arnol'd in the 1970's ("Discoveries are rarely attributed to the correct person."), Carl Boyer in 1968 ("Clio, the muse of history, often is fickle in attaching names to theorems!"), Alfred North Whitehead in 1917 ("Everything of importance has been said before by someone who did not discover it."), and even Stephen's father George Stigler in 1966 ("If we should ever encounter a case where a theory is named for the correct man, it will be noted.")! The law was dubbed the Zeroth Theorem of the History of Science by historian Ernst Peter Fischer in 2006 ("[E]ine Entdeckung (Regel, Gesetzmässigkeit, Einsicht), die nach einer Person benannt ist, nicht von dieser Person herrührt.") .

in doctor α's preference list, and $HosPref[A][r]$ represents the rth doctor in hospital A's preference list. With the input in this form, the Boston Pool algorithm can be implemented to run in $O(n^2)$ time; we leave the details as an exercise for the reader. A somewhat harder exercise is to prove that there are inputs (and choices of who makes offers when) that force $\Omega(n^2)$ rounds before the algorithm terminates.

But why is it *correct*? Gale and Shapely prove that the Boston Pool algorithm always computes a stable matching as follows. The algorithm must terminate, because each hospital makes an offer to each doctor at most once. When the algorithm terminates, every internship has been filled. Now suppose in the final matching, doctor α is assigned to hospital A but prefers B. Since every doctor accepts the best offer she receives, α received no offer she liked more than A. In particular, B never made an offer to α. On the other hand, B made offers to every doctor they like more than β. Thus, B prefers β to α, and so there is no instability.

Surprisingly, the correctness of the algorithm does not depend on which hospital makes its offer in which round. In fact, there is a stronger sense in which the order of offers doesn't matter—no matter which unassigned hospital makes an offer in each round, *the algorithm always computes the same matching*! Let's say that α is a *feasible* doctor for A if there is a stable matching that assigns doctor α to hospital A, and let $best(A)$ be the highest-ranked feasible doctor on A's preference list.

Lemma 1. *The Boston Pool algorithm assigns $best(A)$ to A, for every hospital A.*

Proof: In the final matching, no hospital A is assigned a doctor they prefer to $best(A)$, because that would create an instability (by definition of 'feasible' and 'best'). A hospital A can only be assigned a doctor they like less than $best(A)$ if $best(A)$ rejects their offer.

So consider the *first* round where some hospital A is rejected by its best choice $\alpha = best(A)$. In that round, α got an offer from another hospital B that α ranks higher than A. Now, B must rank α at least as highly as its best choice $best(B)$, because this is the *first* round where a hospital is rejected by its best choice. Thus, B must rank α higher than *any* doctor that is feasible for B.

Now consider the stable matching where α is assigned to A. On the one hand, α prefers B to A. On the other hand, because this is a stable matching, B is assigned a doctor β that is feasible for B, which implies that B prefers α to β. But this means the matching is unstable, and we have a contradiction. □

In other words, from the hospitals' point of view, the Boston Pool algorithm computes the best possible stable matching. It turns out that this is also the *worst* possible matching from the doctors' viewpoint! Let $worst(\alpha)$ be the lowest-ranked feasible hospital on doctor α's preference list.

Lemma 2. *The Boston Pool algorithm assigns α to $worst(\alpha)$, for every doctor α.*

Proof: Suppose the Boston Pool algorithm assigns doctor α to hospital A; the previous lemma implies that $\alpha = best(A)$. To prove the lemma, we need to show that $A = worst(\alpha)$.

Consider an arbitrary stable matching where A is *not* matched with α but with another doctor β. Then A must prefer α to β. Since this matching is stable, α must therefore prefer her current assignment to A. This argument works for *any* stable assignment, so α prefers *every* other feasible match to A; in other words, $A = worst(\alpha)$. □

In 1998, the National Residency Matching Program reversed its matching algorithm, so that potential residents offer to work for hospitals, and each hospital accepts its best offer. Thus, the new algorithm computes the best possible stable matching for the doctors, and the worst possible stable matching for the hospitals. As far as I know, the precise effect of this change on the *patients* is an open problem.

0.6 Why are we here, anyway?

This class is ultimately about learning two skills that are crucial for all computer scientists.

1. ***Reasoning:*** How to *think* about abstract computation.

2. ***Communication:*** How to *talk* about abstract computation.

The first 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? These are *not* easy questions. Anyone who says differently is selling something.

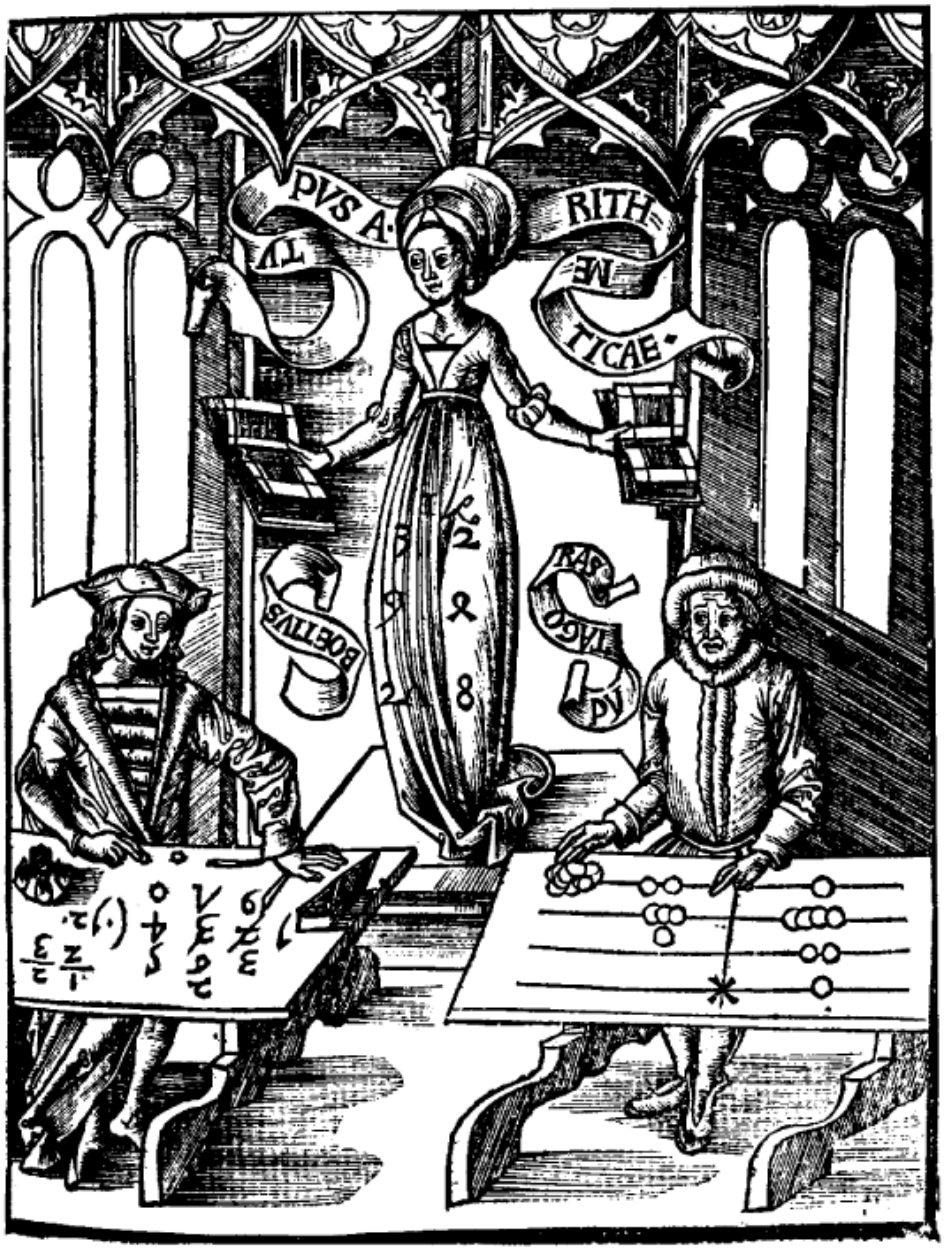
Our second main goal is to help you develop algorithmic *language*. It's not enough just to understand how to solve a problem; you also have to be able to explain your solution to somebody else. I don't mean just how to turn your algorithms into working code—despite what many students (and inexperienced 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 to clarify your own understanding. As Albert Einstein (or was it Richard Feynman?) apocryphally put it, "You do not really understand something unless you can explain it to your grandmother."

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 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 *incredibly* useful, and it's impossible to develop good intuition and good communication skills without them, but they aren't the main point of the course either. At this point in your educational career, you should be able to pick up most of those skills on your own, once you know what you're trying to do.

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 activities (music, writing, juggling, acting, martial arts, sports, cooking, programming, teaching, etc.), the *only* way to master these skills is to make them your own, through practice, practice, and more practice. You can only develop good problem-solving skills by solving problems. You can only develop good communication skills by communicating. Good intuition is the product of experience, not its replacement. We *can't* teach you how to do well in this class. All we can do (and what we will do) is lay out some fundamental tools, show you how to use them, create opportunities for you to practice with them, and give you honest feedback, based on our own hard-won 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!



Boethius the algorist versus Pythagoras the abacist.
from *Margarita Philosophica* by Gregor Reisch (1503)

Exercises

0. Describe and analyze an algorithm that determines, given a legal arrangement of standard pieces on a standard chess board, which player will win at chess from the given starting position if both players play perfectly. *[Hint: There is a one-line solution!]*

1. The traditional Devonian/Cornish drinking song "The Barley Mow" has the following pseudolyrics[18], where *container*$[i]$ is the name of a container that holds 2^i ounces of beer. One version of the song uses the following containers: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. (Every container in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.)

```
BARLEYMOW(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the container[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The container[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

   (a) Suppose each container name *container*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)

   (b) If you want to sing this song for $n > 20$, you'll have to make up your own container names. To avoid repetition, these names will get progressively longer as n increases[19]. Suppose *container*$[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)

   (c) Suppose each time you mention the name of a container, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *container*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BARLEYMOW(n)? (Give an *exact* answer, not just an asymptotic bound.)

[18] Pseudolyrics are to lyrics as pseudocode is to code.

[19] "We'll drink it out of the hemisemidemiyottapint, boys!"

2. Recall that the input to the Huntington-Hill apportionment algorithm ApportionCongress is an array $P[1..n]$, where $P[i]$ is the population of the ith state, and an integer R, the total number of representatives to be allotted. The output is an array $r[1..n]$, where $r[i]$ is the number of representatives allotted to the ith state by the algorithm.

   Let $P = \sum_{i=1}^{n} P[i]$ denote the total population of the country, and let $r_i^* = P[i] \cdot R/P$ denote the ideal number of representatives for the ith state.

   (a) Prove that $r[i] \geq \lfloor r_i^* \rfloor$ for all i.

   (b) Describe and analyze an algorithm that computes exactly the same congressional apportionment as ApportionCongress in $O(n \log n)$ time. (Recall that the running time of ApportionCongress depends on R, which could be arbitrarily larger than n.)

   (c) If a state's population is small enough relative to the other states, its ideal number r_i^* of representatives could be very close to zero; thus, tiny states are over-represented by the Huntington-Hill apportionment process. Surprisingly, this can also be true of very large states. Let $\alpha = (1 + \sqrt{2})/2 \approx 1.20710678119$. Prove that for any $\varepsilon > 0$, there is an input to ApportionCongress with $\max_i P[i] = P[1]$, such that $r[1] > (\alpha - \varepsilon)\, r_1^*$.

   ★(d) Can you improve the constant α in the previous question?

3. Describe and analyze the Boston Pool stable matching algorithm in more detail, so that the worst-case running time is $O(n^2)$.

4. Consider a generalization of the stable matching problem, where some doctors do not rank all hospitals and some hospitals do not rank all doctors, and a doctor can be assigned to a hospital only if each appears in the other's preference list. In this case, there are three additional unstable situations:

   - A hospital prefers an unmatched doctor to its assigned match.
   - A doctor prefers an unmatched hospital to its assigned match.
   - An unmatched doctor and an unmatched hospital each appear in the other's preference list.

   Describe and analyze an efficient algorithm that computes a stable matching in this setting.

   Note that a stable matching may leave some doctors and hospitals unmatched, even though their preference lists are non-empty. For example, if every doctor lists Harvard as their only acceptable hospital, and every hospital lists Doogie as their only acceptable intern, then only Doogie and Harvard will be matched.