Hinc incipit algorismus. *Haec algorismus ars praesens dicitur in qua talibus indorum fruimur bis quinque figuris 0. 9. 8. 7. 6. 5. 4. 3. 2. 1.*

— Friar Alexander de Villa Dei, *Carmen de Algorismo*, (c. 1220)

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.*

— Augusta Ada King-Noel, née Byron, 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 moment a man begins to talk about technique that's proof that he is fresh out of ideas.

— Raymond Chandler

Chapter 0

Introduction

1.1 What is an algorithm?

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions, usually intended to accomplish a specific purpose. 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 roots *arithmos* (άριθμός), meaning "number", and *algos* (ἄλγος), meaning "pain". Rather, it is a corruption of the name of the 9th century Persian mathematician

Muḥammad ibn Mūsā al-Khwārizmī (محمد بن موسى الخوارزمي).[1] Al-Khwārizmī is perhaps best known as the writer of the treatise *Al-Kitāb al-mukhtaṣar fīhīsāb al-ğabr wa'l-muqābala* (الكتاب المختصر في حساب الجبر والمقابلة)[2], from which the modern word *algebra* derives. In a different treatise, al-Khwārizmī described the modern decimal system for writing and manipulating numbers—in particular, the use of a small circle or *ṣifr* (صفر) to represent a missing quantity—which had been developed in India several centuries earlier. This system later became known in Europe as ***algorism***, and its figures became known in English as *ciphers*.[3]

The decimal place-value system was popularized in Europe by the medieval Italian mathematician and tradesman Leonardo of Pisa, better known as Fibonacci. Thanks in part to his 1202 book *Liber Abaci*, written figures began to replace the counting table (then known as an *abacus*) as the preferred platform for calculation in Europe in the early 13th century.[4] However, ciphers became common in Western Europe only with the advent of movable type, and truly ubiquitous only after cheap paper became plentiful in the early 19th century.

The more modern word *algorithm* is a false cognate with the Greek word *arithmos* (and perhaps the previously mentioned *algos*).[5] Thus, until very recently, the word *algorithm* referred exclusively to pencil-and-paper methods for numerical calculations. People trained in the fast and reliable execution of these procedures were called—you guessed it—*computers*.

1.2 Multiplication

Although they have been a topic of formal academic study for only a few decades, algorithms have been with us since the dawn of civilization. Descriptions of step-by-step arithmetic computation are among the earliest examples of written human language.

[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]Fibonacci transliterated صفر as *zephirum* in Latin, which became *zefiro* in Italian, which later evolved into the modern *zero*.

[4]The word *calculate* derives from the Latin word *calculus*, meaning "small rock", referring to the stones on a counting table. While it is tempting to translate the title *Liber Abaci* as "The Book of the Abacus", a more accurate translation is "The Book of Calculation". Both before and after Fibonacci, the Italian word *abaco* was used to describe anything related to numerical calculation—devices, methods, schools, books, and so on—much in the same way that "computer science" is used today in English, or as the Chinese phrase for "operations research" translates literally as "the study of using counting rods".

[5]Some medieval sources claimed that the Greek prefix "algo-" meant "art" or "introduction". Others claimed that algorithms was invented by a Greek philosopher, or a king of India, or perhaps a king of Spain, named "Algus" or "Algor" or "Argus". A few, possibly including Dante Alighieri, even identified the inventor with the mythological Greek shipbuilder and eponymous argonaut. It's unclear whether these risible claims were intended to be historical or merely mnemonic.

Lattice Multiplication

The most familiar method for multiplying large numbers, at least for American students, is the ***lattice algorithm***. This algorithm was introduced to Europeans by Fibonacci in *Liber Abaci*, who learned it from Arabic sources including al-Khwārizmī, who in turn learned it from Indian sources including Brahmagupta's 7th-century treatise *Brāhmasphuṭasiddhānta*, who most likely learned it from Chinese sources. The oldest surviving detailed descriptions of the algorithm are in *The Mathematical Classic of Sunzi*, written in China in the 5th century.[6] However, there is some evidence that the lattice algorithm was known much earlier; the Sumerians recorded multiplication tables on clay tablets as early as 2600BC.

The lattice algorithm assumes that the input numbers are represented as explicit strings of digits; I'll assume here that we're working in base ten, but the algorithm generalizes immediately to any other base. The input to the algorithm is a pair of arrays $X[0\,..\,m-1]$ and $Y[1\,..\,n-1]$, representing the numbers

$$x = \sum_{i=0}^{m-1} X[i]\cdot 10^i \quad \text{and} \quad y = \sum_{j=0}^{n-1} Y[j]\cdot 10^j.$$

Similarly, the output is an array $Z[0\,..\,m+n-1]$ representing the product

$$z = x\cdot y = \sum_{k=0}^{m+n-1} Z[k]\cdot 10^k.$$

The algorithm uses *single-digit* multiplication as a primitive operation; in practice, this operation is performed using a lookup table, either carved into clay tablets, painted on strips of wood or bamboo, written on paper, stored in read-only memory, or memorized by the computer. In fact, the entire algorithm can be summarized by the formula

$$x\cdot y \;=\; \sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\bigl(X[i]\cdot Y[j]\cdot 10^{i+j}\bigr).$$

Different variants of the lattice algorithm evaluate the single-digit partial products $X[i]\cdot Y[j]\cdot 10^{i+j}$ in different orders and use different strategies for computing their sum. For example, in *Liber Abaco*, Fibonacci describes a variant that considers the mn partial products in increasing order of significance, as shown in Figure 1.1.

Fibonacci's algorithm is often executed by storing all the partial products in a two-dimensional table (often called a "tableau" or "lattice") and then summing along the diagonals with appropriate carries, as shown on the right in Figure 1.2. American elementary-school students are taught to multiply one factor by each digit in the other factor, writing down all the partial products before adding them up, as shown on the left

[6] *The Art of War* was written many centuries earlier by a different Sun Zi.

```
FibonacciMultiply(X[0..m−1], Y[0..n−1]):
  hold ← 0
  for k ← 0 to n+m−1
      for all i and j such that i+j = k
          hold ← sum + X[i]·Y[j]
      Z[k] ← hold mod 10
      hold ← ⌊hold/10⌋
  return Z[0..m+n−1]
```

Figure 1.1. Fibonacci's multiplication algorithm, from *Liber Abaco* (1202)

in Figure 1.2. Both of these variants (and several others) are described and illustrated side by side in the anonymous 1458 textbook *L'Arte dell'Abbaco*, also known as the *Treviso Arithmetic*, the very first printed mathematics book in the West.

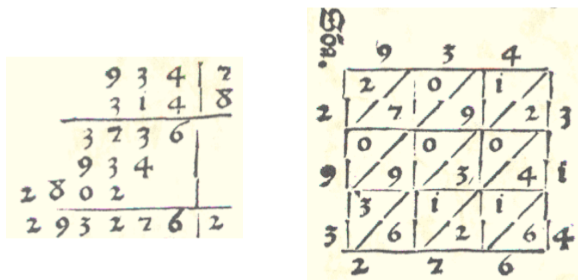

Figure 1.2. Computing $934 \times 314 = 293276$ using "long" multiplication (with error-checking by casting out nines) and "lattice" multiplication, from *L'Arte dell'Abbaco* (1458).

Both of these variants of the lattice algorithm—and related variants described by Sunzi, al-Khwārizmī, *L'Arte dell'Abbaco*, and many other sources—run in $\Theta(mn)$ time; in every variant, the running time is dominated by the number of single-digit multiplications.

Duplation and Mediation

The lattice algorithm is not the oldest multiplication algorithm for which we have direct recorded evidence. An even older and arguably simpler algorithm, which does not rely on a place-value representation at all, is sometimes called *Russian peasant multiplication*, *Ethiopian peasant multiplication*, or just ***peasant multiplication***. A variant of this algorithm was copied into the Rhind papyrus by the Egyptian scribe Ahmes around 1650BC, from a document he claimed was (then) about 350 years old. This algorithm was still taught in elementary schools in Eastern Europe in the late 20th century; it was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.

The peasant multiplication algorithm breaks the difficult task of multiplying arbitrary numbers into four simpler operations: (1) determining parity (even or odd), (2) addition, (3) ***duplation*** (doubling a number), and (4) ***mediation*** (halving a number, rounding

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

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

Figure 1.3. Multiplication by duplation and mediation

down).[7] Of course a full specification of this algorithm requires describing how to perform those four primitive operations.

The correctness of peasant multiplication follows by induction from the following simple identity, which holds for all non-negative integers x and y:

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

As stated, the algorithm requires $\lceil \log_2 x \rceil = O(\log x)$ parity, addition, and mediation operations, but we can easily improve this to $O(\log \min\{x, y\})$ by swapping the two arguments when $x > y$. Assuming the numbers are represented using any reasonable place-value notation (like binary, decimal, Babylonian hexagesimal, Egyptian duodecimal, Roman numeral, Chinese counting rods, bead positions on an abacus, and so on), then each operation requires at most $O(\log(xy)) = O(\log \max\{x, y\})$ single-digit additions, so the overall running time of the algorithm is $O(\log \min\{x, y\} \cdot \log \max\{x, y\}) = O(\log x \cdot \log y)$.

In other words, this algorithm requires $O(mn)$ time to multiply an m-digit number by an n-digit number. This algorithm requires (a constant factor!) more paperwork to execute by hand than the lattice algorithm, but the necessary primitive operations are arguably easier for humans to perform. In fact, the two algorithms are equivalent when numbers are represented in binary.

Compass and Straightedge

Classical Greek geometers identified numbers (or more accurately, magnitudes) with line segments of the appropriate length, which they manipulated through two simple

[7]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.

mechanical tools: the compass and the straightedge. These tools could be used to construct points, lines, and circles using any sequence of the following primitive operations:

- Draw the unique line between two distinct points.
- Draw the unique circle centered at one point and passing through another points.
- Find the intersection point of two non-parallel lines.
- Find the intersection points (if any) between a line and a circle.
- Find the intersection points (if any) between two circles.

In practice, Greek geometry students almost certainly drew their constructions on an *abax* (ἄβαξ), a table covered in dust or sand.[8] However, Euclid and other Greek geometers wrote about compass and straightedge constructions as precise mathematical abstractions—points are ideal points; lines are ideal lines; and circles are ideal circles.

The following algorithm for multiplying or dividing numbers was popularized (but almost certainly not discovered) by Euclid about 2500 years ago. The input consists of four distinct points A, B, C, D, and the goal is to construct a point Z such that $|AZ| = |AC||AD|/|AB|$. In particular, if we take $|AB|$ to be the unit length, then the algorithm computes the product of $|AC|$ and $|AD|$. Notice that Euclid first defines a new primitive operation RIGHTANGLE by (as modern programmers would phrase it) writing a subroutine.

```
⟨⟨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))
```

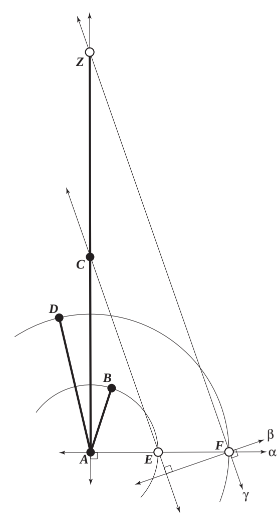

Figure 1.4. Multiplying or dividing using a compass and straightedge.

This algorithm reduces the problem of multiplying two numbers (lengths) into a series of primitive ruler-and-compass operations. These operations are incredibly

[8]The numerals 1 through 9 were known in Europe at least two centuries before Fibonacci's *Liber Abaci* as "gobar numerals", from the Arabic word *ghubār* (غبار) meaning dust, ultimately referring to the Indian practice of performing arithmetic on tables covered with sand. The Greek word *abax* is the origin of the Latin *abacus*, which also originally referred to a sand table.

difficult to implement precisely 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, for whom each operation can be performed perfectly in constant time *by definition*. In this model of computation, the compass-and-straightedge multiplication algorithm runs in $O(1)$ "time".[9]

1.3 Congressional Apportionment

Here is another real-world example of an algorithm of significant political importance. Article I, Section 2 of the United States 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. . . .

Because there are only a finite number of seats in the House of Representatives, exact proportional representation is impossible without either shared or fractional representatives, neither of which are legal. As a result, over the next several decades, several different apportionment algorithms were 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.[10] The input array $Pop[1 .. n]$ stores the populations of the n states, and R is the total number of representatives. Currently, $n = 50$ and $R = 435$. The output array $Rep[1 .. n]$ stores the number of representatives assigned to each state.

This pseudocode description assumes a priority queue that supports the operations NewPriorityQueue, Insert, and ExtractMax. (The actual law doesn't assume that,

[9] Of course, Greek tradesmen never used this approach to multiply actual numbers for actual real-world bookkeeping. Greek scholars maintained a clear distinction between *arithmetic*, which we now call number theory, and *logistic* (λογιστικός), which described practical numerical computation. Logistic was considered a purely practical skill for tradesmen, similar to typing in the 20th century, unworthy of serious scholarly attention. Perhaps as a result of this intellectual snobbery, there is no surviving record of Greek logistic algorithms, aside from a few counting boards, multiplication tables, and a passing references to reckoning with *psephoi* (ψῆφοι), literally "pebbles". For example, in 440BC, Herodotus wrote in his *Histories* that "The Greeks write and calculate (λογίζονται ψήφοισι) from left to right; the Egyptians do the opposite. Yet they say that their way of writing is toward the right, and the Greek way toward the left."

[10] 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.

```
ApportionCongress(Pop[1..n], R):
  PQ ← NewPriorityQueue
  for i ← 1 to n
      Rep[i] ← 1
      Insert(PQ, i, Pop[i]/√2)
      R ← R − 1

  while R > 0
      s ← ExtractMax(PQ)
      Rep[s] ← Rep[s] + 1
      Insert(PQ, s, Pop[s]/√(Rep[s](Rep[s] + 1)))
      R ← R − 1

  return Rep[1..n]
```

Figure 1.5. The Huntington-Hill apportionment algorithm

of course.) The output of the algorithm, and therefore its correctness, does not depend *at all* on how this priority queue is implemented. The Census Bureau uses 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.

Similar apportionment algorithms are in common use around the world, in parliamentary elections where the number of seats allocated to each political party is supposed to be proportional to the number of votes that party receives. However, the Huntington-Hill method is essentially unique to the United States House of Representatives, thanks to the constitutional requirement that each state has at least one representative. The most common apportionment algorithms in the rest of the world, called the ***D'Hondt method*** and the ***Webster/Sainte-Laguë method***, use priorities $Pop[s]/(Rep[s]+1)$ and $Pop[s]/(2 \cdot Rep[s]+1)$, respectively, in place of the square-root expression used by Huntington-Hill.

1.4 A Bad Example

As a prototypical example of a sequence of instructions that is not actually an algorithm, consider Martin's "algorithm":[11]

```
BecomeAMillionaireAndNeverPayTaxes:
  Get a million dollars.
  If the tax man comes to the door and says, "You have never paid taxes!"
      Say "I forgot."
```

Pretty simple, except for that first step; it's a doozy. A group of billionaire CEOs (or Silicon Valley venture capitalists) might consider this an algorithm, since for them

[11] Steve Martin, "You Can Be A Millionaire", Saturday Night Live, January 21, 1978. Also appears on *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

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 actual 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 some 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 reasonably implemented on a standard digital 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).

1.5 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. The idiosyncratic syntactic details of C, C++, C#, Java, Python, Ruby, Erlang, Haskell, OcaML, Scheme, Scala, Clojure, Visual Basic, Smalltalk, Javascript, Processing, Squeak, Forth, TeX, Fortran, COBOL, INTERCAL, MMIX, LOLCODE, Befunge, Parseltongue, Whitespace, or Brainfuck are irrelevant in algorithm design; focusing on them will only distract you from what's *really* going on.[12] 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 lots of structure—especially conditionals, loops, and recursion—that are far too easily hidden by unstructured prose. Natural languages like English are full of ambiguities, subtleties, and shades of meaning, but algorithms must be described as precisely and unambiguously as possible. More seriously, in non-technical writing,

[12]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. For a compelling counterexample, see Chris Okasaki's thesis/monograph *Functional Data Structures* and its more recent descendants.

there is natural tendency to describe repeated operations informally: "Do this first, then do this second, and so on." But 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 says little or nothing about what happens in later iterations. To make the description unambiguous, we must explicitly specify the behavior of *every* iteration. The stupid joke about the programmer dying in the shower has a grain of truth—"Lather, rinse, repeat" *is* ambiguous; what exactly do we repeat, and until when?

In my opinion, pseudocode is the clearest way to present most algorithms. Pseudocode uses the *structure* of formal programming languages and mathematics to break algorithms into primitive steps; but the primitive steps themselves may be written using mathematics, pure English, or an appropriate mixture of the two. Well-written pseudocode reveals the internal structure of the algorithm but hides irrelevant implementation details, making the algorithm much easier to understand, analyze, debug, implement, and generalize.

The precise syntax of pseudocode is a personal choice, but the overriding goal should be clarity and precision. Ideally, pseudocode should allow any competent programmer to implement the underlying algorithm, quickly and correctly, in *their* favorite programming language, *without understanding why the algorithm works*. Here are the guidelines I follow and strongly recommend:

- Use standard imperative programming keywords (if/then/else, while, for, return, and the like) and standard notation, like $array[index]$ for array indexing, $function(argument)$ for function calls, and so on. Use the syntax $variable \leftarrow value$ for assignment, to avoid the = versus == (versus ===) problem. Keywords should always be standard English words: always write "else if" instead of "elif".

- Indent everything carefully and consistently; the block structure should be visible from across the room. This rule is especially important for nested loops and conditionals. *Don't* add unnecessary syntactic sugar like braces or begin/end tags; careful indentation is almost always enough. In other words, indent as though you're writing in Python.

- Use mnemonic algorithm and variable names. Short variable names are good, but readability is more important than brevity; except for idioms like loop indices, short but complete words are better than single letters or opaque abbrvtns. Absolutely *never* use pronouns!

- Use standard mathematical notation for standard mathematical things. For example, write $x \cdot y$ instead of $x * y$ for multiplication; $x = y$ for equality; $x \geq y$ instead of $x >= y$ for weak comparison; $x \bmod y$ instead of $x \,\%\, y$ for remainder; $\sqrt{x}$ instead of $sqrt(x)$ for square root; a^b instead of $power(a, b)$ or $a ** b$ for exponentiation; and ϕ instead of phi for the golden ratio.

- Avoid mathematical notation if English is clearer. For example, "Insert a into X" may be preferable to "$\textsc{Insert}(X, a)$" or "$X \leftarrow X \cup \{a\}$".

- 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 $i \leftarrow i+1;\ j \leftarrow j-1;\ k \leftarrow 0$.)
- Don't typeset pseudocode in a `fixed-width typeface`; it's much harder to read than normal text. Don't typeset keywords like "for" or "while" in bold; **the** syntactic sugar **is** not **what** you want **the** reader **to** look **at**. On the other hand, I do use *italics* for variables (following standard mathematical typesetting conventions), SMALL CAPS for algorithms and constants (like TRUE, FALSE, and NULL), and a different typeface for literal strings.

1.6 Analyzing algorithms

It's not enough just to write down an algorithm and say 'Behold!' We must also convince our audience (and ourselves!) that the algorithm actually does what it's supposed to do, and that it does so 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 our algorithms are correct; trusting our instincts, or trying a few test cases, isn't good enough. 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 extra work to prove correct. Correctness proofs almost always involve induction. We *like* induction. Induction is our *friend*.[13]

But before we can formally prove that our algorithm does what it's supposed to do, we have to formally *state* what it's supposed to do! Algorithmic problems are often presented using standard English, in terms of real-world objects, not in terms of formal mathematical objects. 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 must also determine if the problem statement carries 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.[14] Restating the problem formally is not only required for proofs; it is also one of the best ways to really understand what a problem is asking for. The hardest part of answering any question is figuring out the right way to ask it!

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

[14] Occasionally one can hear set theorists singing "$\aleph_0$ bottles of beer on the wall, $\aleph_0$ bottles of beer! Take one down, pass it around, $\aleph_0$ bottles of beer on the wall!", but they always give up before the song is over.

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 for accomplishing such a task. The same problem may have hundreds of different algorithms; the same algorithm may solve hundreds of different problems.

Running time

The most common way of ranking different algorithms for the same problem is by how quickly they run. Ideally, we want the fastest possible algorithm for any particular 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($2n$) requires about twice much time 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) mentions beer exactly $3n+3$ times.

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.

There are many other traditional songs that take quadratic time to sing; examples include "There Was an Old Lady Who Swallowed a Fly", "The House that Jack Built", "Hole in the Bottom of the Sea", "The Green Grass Grew All Around", "Green Grow the Rushes O", "The Rattlin' Bog", "The Court Of King Caractacus", "An Austrian Went Yodeling", "Hey, Hey, Bo Diddly Bop", "The Barley-Mow", "Eh, Cumpari!", "I have a Song to Sing, O!", "Star Trekkin'", "Echad Mi Yode'a", "Ist das nicht ein Schnitzelbank?", and "Minkurinn í hænsnakofanum". For further details, consult your nearest preschooler or Boy Scout. A more modern example is "The TELNET Song" by Guy Steele, which takes $\Theta(2^n)$ time to sing the first n verses; Steele recommended $n = 4$.

```
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."
```

```
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 « Alouette, alouette, ooooooo! »
      Chantez « Alouette, gentille allouette, alouette, je te plumerais. »
```

Figure 1.6. Eight brass monkeys from the ancient, sacred, holy crypts of Egypt.

For a slightly less facetious example, consider the algorithm ApportionCongress. Here the running time obviously depends on the implementation of the priority queue operations, but we can certainly bound the running time as $O(N + RI + (R - n)E)$, where N denotes the running time of NewPriorityQueue, I denotes the running time of Insert, and E denotes the running time of ExtractMax. Under the reasonable assumption that $R > 2n$ (on average, each state gets at least two representatives), we can simplify the bound to $O(N + R(I + E))$. The Census Bureau implements the priority queue using an unsorted array of size n; this implementation gives us $N = I = \Theta(1)$ and $E = \Theta(n)$, so the overall running time is $O(Rn)$. This is good enough for government work, but we can do better. Implementing the priority queue using a binary heap (or a heap-ordered array) gives us $N = \Theta(1)$ and $I = R = O(\log n)$, which implies an overall running time of $O(R \log n)$.

Sometimes we are also interested in other computational resources: space, randomness, page faults, inter-process messages, and so forth. We can use the same techniques to analyze those resources as we use to analyze running time. For example, lattice multiplication of two n-digit numbers requires $\Theta(n^2)$ space if we write down all the partial products before adding them, but only $O(n)$ space if we add them on the fly.

1.7 A Longer Example: Stable Matching

★★★

Arguably this fits better under greedy algorithms.

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 internships, and demand *immediate* responses. 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.[15]

Finally, a central clearinghouse for internship assignments, now called the National Resident Matching Program (NMRP), 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

[15]The academic job market involves similar gambles, at least in computer science. Some departments start making offers in February with two-week decision deadlines; other departments don't even start interviewing until March; MIT notoriously waits until May, when all its interviews are over, before making *any* faculty offers.

[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.

hospital has an incentive to cheat the system. At first glance, it is not clear that a stable matching exists!

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 misattributed to David Gale and Lloyd Shapley, who formally analyzed the algorithm and first proved that it computes a stable matching in 1962; Gale and Shapley used the metaphor of college admissions.[17] Similar algorithms have since been adopted for other matching markets, including faculty recruiting in France, university admission in Germany, public school admission in New York and Boston, billet assignments for US Navy sailors, and kidney-matching programs. Shapley was awarded the 2012 Nobel Prize in Economics for his research on stable matching, together with Alvin Roth, who significantly extended Shapley's work and used it to develop several real-world exchanges.

The Boston Pool algorithm proceeds in rounds until every position has been filled. Each round has two stages:

1. An arbitrary unassigned hospital A offers its position to the best doctor α (according to the hospital's preference list) who has not already rejected it.
2. Each doctor ultimately accepts the best offer that she receives, according to her preference list. Thus, if α is currently unassigned, she (tentatively) accepts the offer from A. If α already has an assignment but prefers A, she rejects her existing assignment and (tentatively) accepts the new offer from A. Otherwise, α rejects the new offer.

For example, suppose four doctors (Dr. Quincy, Dr. Rotwang, Dr. Shephard, and Dr. Tam, represented by lower-case letters) and four hospitals (Arkham Asylum, Bethlem Royal Hospital, County General Hospital, and The Dharma Initiative, represented by upper-case letters) rank each other as follows:

q	r	s	t
A	A	B	D
B	D	A	B
C	C	C	C
D	B	D	A

A	B	C	D
t	r	t	s
s	t	r	r
r	q	s	q
q	s	q	t

Given these preferences as input, the Boston Pool algorithm might proceed as follows:

[17]The "Gale-Shapley 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. However, similar statements were previously made by 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."). We will see *many* other examples of Stigler's law in this class.

1. Arkham makes an offer to Dr. Tam.
2. Bedlam makes an offer to Dr. Rotwang.
3. County makes an offer to Dr. Tam, who rejects her earlier offer from Arkham.
4. Dharma makes an offer to Dr. Shephard. (From this point on, because there is only one unmatched hospital, the algorithm has no more choices.)
5. Arkham makes an offer to Dr. Shephard, who rejects her earlier offer from Dharma.
6. Dharma makes an offer to Dr. Rotwang, who rejects her earlier offer from Bedlam.
7. Bedlam makes an offer to Dr. Tam, who rejects her earlier offer from County.
8. County makes an offer to Dr. Rotwang, who rejects it.
9. County makes an offer to Dr. Shephard, who rejects it.
10. County makes an offer to Dr. Quincy.

At this point, all pending offers are accepted, and the algorithm terminates with a matching: (A,s), (B,t), (C,q), (D,r). You can (and should) verify by brute force that this matching is stable, even though no doctor was hired by her favorite hospital, and no hospital hired its favorite doctor; in fact, County was forced to hire their *least* favorite doctor. This is not the only stable matching for this list of preferences; the matching $(A,r),(B,s),(C,q),(D,t)$ is also stable.

Running Time

Analyzing the algorithm's running time is relatively straightforward. Each hospital makes an offer to each doctor at most once, so 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 $\mathit{DocPref}[1..n][1..n]$ and $\mathit{HosPref}[1..n][1..n]$, where $\mathit{DocPref}[\alpha][r]$ represents the rth hospital in doctor α's preference list, and $\mathit{HosPref}[A][r]$ 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 easy exercise.

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. Thus, the $O(n^2)$ upper bound on the worst-case running time cannot be improved; in this case, we say our analysis is ***tight***.

Correctness

But why is the algorithm *correct*? How do we know that the Boston Pool algorithm always computes a *stable matching*? Gale and Shapley proved correctness as follows. The algorithm continues as long as there is at least one unfilled position; conversely, when the algorithm terminates (after at most n^2 rounds), every position is filled. No doctor can accept more than one position, and no hospital can hire more than one doctor. Thus, the algorithm always computes a matching; it remains only to prove that the matching is stable.

Suppose doctor α is assigned to hospital A in the final matching, but prefers B. Because 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.

Lemma 0.1. *During the Boston Pool algorithm, each hospital A is rejected only by doctors that are infeasible for A.*

Proof: We prove the lemma by induction. Consider an arbitrary round of the Boston Pool algorithm, in which doctor α rejects one hospital A for another hospital B. The rejection implies that α prefers B to A. Every doctor that appears higher than α in B's preference list has already rejected B and therefore, by the inductive hypothesis, is infeasible for B.

Now consider an arbitrary matching that assigns α to A. We already established that α prefers B to A. If B prefers α to its partner, the matching is unstable. On the other hand, if B prefers its partner to α, then (by our earlier argument) its partner is infeasible, and again the matching is unstable. We conclude that there is no stable matching that assigns α to A. □

Now let ***best*(*A*)** denote the highest-ranked *feasible* doctor on A's preference list. Lemma 0.1 implies that every doctor that A prefers to its final assignment is infeasible for A. On the other hand, the final matching is stable, so the doctor assigned to A is feasible for A. The following result is now immediate:

Corollary 0.2. *The Boston Pool algorithm assigns* best(A) *to A, for every hospital A.*

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

Corollary 0.3. *The Boston Pool algorithm assigns α to* worst(α)*, for every doctor α.*

Proof: Suppose the Boston Pool algorithm assigns doctor α to hospital A; we need to show that $A = \mathit{worst}(\alpha)$. Consider an arbitrary stable matching where A is *not* matched with α but with another doctor β. The previous corollary implies that A prefers

$\alpha = best(A)$ to β. Because the matching is stable, α must therefore prefer her assigned hopital to A. This argument works for *any* stable assignment, so α prefers *every* other feasible match to A; in other words, $A = worst(\alpha)$. □

A subtle consequence of these two corollaries, discovered by Dubins and Freeman in 1981, is that a doctor can potentially improve her assignment by lying about her preferences, but a hospital cannot. (However, a set of hospitals can collude so that *some* of their assignments improve.) Partly for this reason, the National Residency Matching Program reversed its matching algorithm in 1998, so that potential residents offer to work for hospitals in preference order, 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. In practice, however, this modification affected less than 1% of the resident's assignments. As far as I know, the precise effect of this change on the *patients* is an open problem.

1.8 Why are we here, anyway?

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

1. ***Intuition:*** How to *think* about abstract computation.
2. ***Language:*** 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 examples and counterexamples, 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 you will enjoy playing with them as much as I do.

Exercises

0. Describe and analyze an efficient 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 trivial one-line solution!]*

1. (a) Write (or find) a song that requires $\Theta(n^3)$ time to sing the first n verses.
   (b) Write (or find) a song that requires $\Theta(n \log n)$ time to sing the first n verses.
   (c) Write (or find) a song that requires $\Theta(n^{3/2})$ time to sing the first n verses.

2. Careful readers might complain that our analysis of songs like "n Bottles of Beer on the Wall" or "The n Days of Christmas" is overly simplistic, because larger numbers take longer to sing than shorter numbers. We can more accurately estimate singing time by counting the number of *syllables* sung, rather than the number of *words*.
   (a) How long does it take to sing "n Bottles of Beer on the Wall"?

(b) How long does it take to sing "The n Days of Christmas"?

3. The cumulative drinking song "The Barley Mow" has been sung throughout the British Isles for centuries. The song has many variants; Figure 1.7 conatins pseudolyrics for one version traditionally sung in Devon and Cornwall, where *vessel*$[i]$ is the name of a vessel that holds 2^i ounces of beer.[18]

```
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 vessel[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The vessel[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!"
```

Figure 1.7. "The Barley Mow".

(a) Suppose each name *vessel*$[i]$ 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 arbitrarily large values of n, you'll have to make up your own vessel names. To avoid repetition, these names must become progressively longer as n increases. Suppose *vessel*$[n]$ 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 vessel, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces

[18] In practice, the song uses some subset of nipperkin, quarter-gill, half-a-gill, gill, quarter-pint, half-a-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel/kilderkin, barrel, hogshead, pipe/butt, tun, well, river, and ocean. With a few exceptions (especially at the end), every vessel in this list is twice as big as its predecessor. Irish and Scottish versions of the song have slightly different lyrics, and they usually switch to people after "gallon".

An early version of the song entitled "Give us once a drink" appears in the play *Jack Drum's Entertainment (or the Comedie of Pasquill and Katherine)* written by John Marston around 1600. ("Giue vs once a drinke for and the black bole. Sing gentle Butler *bally moy*!") There is some question whether Marston wrote the "high Dutch Song" specifically for the play, whether "bally moy" is a mondegreen for "barley mow" or vice versa, or whether it's actually the same song at all. These discussions are best had over n bottles of beer.

for each *vessel*$[i]$. 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.)

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

5. Prove that it is possible for the Boston Pool algorithm to execute $\Omega(n^2)$ rounds. (You need to describe both a suitable input and a sequence of $\Omega(n^2)$ valid proposals.)

6. Describe and analyze an efficient algorithm to determine whether a given set of hospital and doctor preferences has to a *unique* stable matching.

7. 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 her assigned match.
   - An unmatched doctor and an unmatched hospital appear in each other's preference lists.

   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 Dr. House as their only acceptable intern, then only House and Harvard will be matched.

8. 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 $r_i^* = R \cdot P[i]/P$ 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 relative to the other states, its ideal number r_i^* of representatives could be 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?