On Countability, Enumeration, and How to Think Like a Computer Scientist[1]

Chuck Allison
Utah Valley University
chuck.allison@uvu.edu

Keith Olson
Utah Valley University
olsonke@uvu.edu

Abstract

One of the primary aims of a college education is to foster students' ability to think critically and analytically [1]. How does this apply to students of computer science? Many computer science students struggle to master the elemental techniques of recursion, inferring qualitative patterns from data, and mathematical induction over countably infinite sets. In this paper, we illustrate all of the above while developing pedagogically rich solutions to a common example used in typical CS curricula: enumerating the rational numbers. We go deeper than typical curricula to help students think like computer scientists.

Introduction

In the journey from computer science student to computer scientist, one meets a number of obstacles, key examples of which are mastering recursion and gaining a deeper understanding of infinity. To think recursively is an art that requires much experience to master. While students have some (perhaps not sufficient) opportunity during their undergraduate years to practice this skill, the same students typically have less experience dealing with countably infinite sets, and rarely, if ever, are called upon to develop algorithms that enumerate such sets. Here we present intuitive yet rigorous solutions to the problem of showing that the rational numbers comprise a countably infinite set.

Countably Infinite Sets

A countably infinite set has the same cardinality as the natural numbers. To show that an infinite set is countable, one typically exhibits a bijection between that set and the natural numbers. A consequence of this mapping is that the elements of the set in question can be *enumerated* as a sequence: a_1, a_2, a_3, etc.

The set of all strings over an alphabet, $\Sigma = \{a, b\}$, for instance, can be intuitively enumerated as follows:

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, \dots\}$$

where λ represents the empty string. It is evident what the next string in the sequence is because we have presented the strings in *quasi-lexicographic order* (aka *canonical order*; i.e., we've grouped the strings by length and then alphabetically within each group [2]). This illustration suffices to convince most people that Σ^* is countably infinite, since we will visit every string in the set.

This simple problem presents a convenient opportunity to reinforce computer science principles by going just a little deeper. Can we find a bijective function that maps between the natural numbers and the strings of Σ^*? Finding an explicit mapping helps students to think more like a computer scientist.

One approach (see [3]) is to interpret strings over an alphabet as numerals. The positions of digits in numerals correspond to a power of the radix the number is expressed in. For example, the base-r numeral $(d_{k-1}d_{k-2}\dots d_1 d_0)_r$ has the value expressed by the polynomial $d_{k-1}r^{k-1} + d_{k-2}r^{k-2} + \cdots + d_1 r + d_0$. Following this pattern, we can use the values 1 and 2 for the symbols a and b, respectively (i.e., our coefficients,

[1] This is an expanded version of Allison, C., Olson, K., On Countability, Enumeration, and How To Think Like a Computer Scientist, *The Journal of Computing Sciences in Colleges*, Volume 28, 1, October 2013.

d_i, will have one of the following values: $v(a) = 1, v(b) = 2$), and evaluate the associated polynomial to obtain the ordinal for a given string. For example, for the string *bba*, we obtain

$$v(b)\cdot 2^2 + v(b)\cdot 2 + v(a) = 2\cdot 4 + 2\cdot 2 + 1 = 8 + 4 + 1 = 13$$

Hence, *bba* is the 13th string in the sequence. The empty string, λ, has the ordinal 0.

Mapping the other direction, from an ordinal to its associated string, can be achieved if we can define a function, *succ*(*s*), which calculates the lexicographic successor to the string *s*. This is not difficult; we merely mimic what we typically do when adding 1 to a number by hand, as the following Python code illustrates.

```
sigma = ('a', 'b')                # Our alphabet

def succ(s):
    chars = list(s)               # Convert string to a mutable list
    i = len(s) - 1                # Start with rightmost symbol
    biggest = sigma[-1]           # Last symbol in alphabet (e.g,. "9" for decimal)
    while i >= 0:
        if chars[i] == biggest:
            chars[i] = sigma[0]   # Wrap to first/smallest symbol
            i -= 1                # Move left to preceding digit
        else:
            break                 # Found a non-biggest symbol
    else:                         # Executes only when loop completes
        # Prepend the "smallest symbol," (non-biggest wasn't found)
        chars = [sigma[0]] + chars

    if i >= 0:                    # The string wasn't all "9"s
        # Increment the number in position i
        index = sigma.index(chars[i])
        chars[i] = sigma[index+1]
    return ''.join(chars)         # Return as a string
```

With this function in hand, we can obtain the string for any ordinal with the recurrence, $s_n = succ(s_{n-1})$, $s_0 = \lambda$, which can be rendered in Python as

```
def nth_str(n):
    return "" if n == 0 else succ(nth_str(n-1))
```

Another approach to discovering a bijection comes from observing a pattern in the following data representing our sequence (but starting with $v(\lambda) = 1$ this time):

λ	a	b	aa	ab	ba	bb	aaa	aab	aba	abb	baa	bab	bba	bbb
1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The first row is the sequence of strings, entries in the second row contain the ordinal in binary notation of the corresponding string above it, and the third row contains the corresponding ordinals in decimal. There is a pattern that relates the binary numbers with their associated strings: the bits after the leading 1 have a zero wherever the associated string has an *a*, and a one where there is a *b*. Thus, our bijection can be expressed as follows:

String-to-ordinal:

1. Starting with a given string, form a bit string by mapping *a* to 0 and *b* to 1.
2. Prepend a 1 bit
3. Convert the bit string to decimal

Ordinal-to-string:

1. Convert the ordinal to binary
2. Remove the leading 1
3. Form a string by mapping a 0-bit to *a* and a 1-bit to *b*.

These mappings are clearly one-to-one and onto, and, of course, are inverses of each other. Python implementations of these procedures follow.

```
def s2ord(s):
    bits = '1' + ''.join([str(ord(c)-ord('a')) for c in s])
    return int(bits,2)

def ord2s(n):
    s = bin(n)[3:]  # Skip leading "0b1" returned from bin()
    return ''.join([{'0':'a','1':'b'}[c] for c in s])
```

Inferring and implementing patterns from data as in this example is an important skill in learning to think like a computer scientist.

Enumerating the Positive Rational Numbers

To convince students that the positive rational numbers are countable, we typically follow Cantor's diagonal enumeration technique with an infinite table like the following (see Table 1).

r\c	1	2	3	4	**…**
1	1/1	1/2	1/3	1/4	…
2	2/1	2/2	2/3	2	…
3	3/1	3/2	3/3	3/4	…
4	4/1	4/2	4/3	4/4	…
…	…	…	…	…	…

Table 1 (**r** = row, **c** = column, fraction = $\frac{r}{c}$)

The body of this table holds the fractional representation of each positive rational number many times over (an infinite number of times, in fact). The idea is that if we can arrange these in a sequence, then the rational numbers, being the subset containing the unique numbers in this table, will certainly be countable. At this point we simply traverse each, starting with 1/1, to enumerate all the values. The body of the following table (Table 2) contains the 1-based *ordinals* that indicate our order of traversal along each successive diagonal in turn, in a lower-left-to-upper-right fashion (i.e., $a_1 = \frac{1}{1}, a_2 = \frac{2}{1}, a_3 = \frac{1}{2}, a_4 = \frac{3}{1}, a_5 = \frac{2}{2}, a_6 = \frac{1}{3}, etc.$).

r\c	1	2	3	4…
1	1	3	6	10…
2	2	5	9	14…
3	4	8	13	19…
4	7	12	18	25…
…	…	…	…	…

Table 2 — Ordinals of the enumeration corresponding to the numbers in Table 1

Once again, we have an opportunity to reinforce students' mastery of computer science techniques by discovering an *algorithm* for the enumeration.

To find the *n*th rational number in this sequence, we must find the row, *r*, and column, *c*, for a given ordinal, *n*, which corresponds to the rational number $\frac{r}{c}$. For example, for $n = 8, r(8) = 3$ and $c(8) = 2$, since the 8th number in our enumeration order is $\frac{3}{2}$. We can make progress toward a solution for finding *r* and *c* by noticing that the ordinals in the first row are the so-called *triangular numbers*, which are the positive numbers of the form $\frac{k(k+1)}{2}, k \geq 1$. We then proceed by finding the smallest triangular number, T_k, which equals or exceeds *n*. If $n = T_k$, then $r(n) = 1$ and $c(n) = k$ and we are done (e.g., $a_6 = \frac{1}{3}, 6 = T_3 \rightarrow r(6) = 1, c(6) = 3$). If $n < T_k$, then the row *increases* and the column *decreases* by the same amount as we go backwards *down* the diagonal to find *n*. (e.g., for $n = 5, T_3 - n = 6 - 5 = 1$, so $r(5) = r(6) + 1 = 2, c(5) = c(6) - 1 = 2 \longrightarrow a_5 = \frac{2}{2}$).

Since we can easily determine whether a number is triangular, we can find $r(n)$ by the following recursive algorithm.

$$r(n) = \begin{cases} 1 & n = T_k = \dfrac{k(k+1)}{2} \\ r(n+1)+1 & \text{otherwise} \end{cases}$$

This relationship holds because row numbers obey the formula $r(n+1) = r(n) - 1$. The recursion stops when we encounter the next triangular number at the top of the diagonal.

Likewise, $c(n+1) = c(n) + 1$, suggesting the following recursive formula for column numbers.

$$c(n) = \begin{cases} k & n = T_k = \dfrac{k(k+1)}{2} \\ c(n+1)-1 & \text{otherwise} \end{cases}$$

Combining these with a function to test a number for triangularity gives the following succinct Python implementation yielding the row (numerator) and column (denominator) for the *n*th rational number in the enumeration.

```
def istrinum(n):
    k = int((math.sqrt(8*n+1) - 1)/2.0)     # Solve T[k] for k
    return k if n == k*(k+1)/2 else 0       # 0 => False

def row(n):
    return 1 if istrinum(n) else 1+row(n+1)

def col(n):
    k = istrinum(n)
    return k if k else col(n+1)-1
```

Inverting The Mapping

Since the relationship between *k* and T_k is bijective, it can be inverted. We seek, therefore, a mapping that yields the sequence number, *n*, as a function of the row and column: $n = f(r, c)$.

Approach 1

Looking again at Table 2, observe the following relationships as we compare consecutive horizontal ordinals in the same row, starting with Row 1:

$$f(1, c) = f(1, c-1) + c$$
$$f(2, c) = f(2, c-1) + c + 1$$

$$\ldots$$
$$f(r,c) = f(r,c-1) + c + r - 1$$

Similar relationships hold when comparing consecutive ordinals in the same column:

$$f(r,1) = f(r-1,1) + r - 1$$
$$f(r,2) = f(r-1,2) + r$$
$$\ldots$$
$$f(r,c) = f(r-1,c) + r + c - 2$$

Since $f(1,1) = 1$, we can form a recurrence that will work its way to either the first row or first column, and then eventually to position (1,1), as follows.

$$f(r,c) = \begin{cases} 1, & r = c = 1 \\ f(r-1,1) + r - 1, & c = 1 \\ f(1,c-1) + c, & r = 1 \\ f(r,c-1) + r + c - 1, & \text{otherwise} \end{cases}$$

We could have used $f(r-1,c) + r + c - 2$ for the last line as well. Here is a recursive Python implementation of *f*:

```
def f(r,c):
    if r == 1 and c == 1:
        return 1
    elif c == 1:
        return f(r-1,1) + r-1
    elif r == 1:
        return f(1,c-1) + c
    else:
        return f(r,c-1) + r+c-1      # Or f(r-1,c) + r+c-2
```

Approach 2: A Closed Formula

To find a closed formula for $f(r,c)$, recall that the largest number at the top of the *d*th diagonal, which is also the top of the *d*th column, is $T_d = \frac{d(d+1)}{2}$. The numbers on the next (i.e., $(d+1)$th) diagonal will be the $d+1$ numbers, $\{T_d + 1, T_d + 2, \ldots T_d + d, T_d + d + 1\}$. Note that the summands $1,2,\ldots,d,d+1$ in the elements of this set are precisely the column numbers of the corresponding entries on diagonal $d+1$, so we can say that the entries on diagonal $d+1$ are the numbers $T_d + c, 1 \le c \le d+1$.

Note also that on diagonal *d*, the sum of the row and column numbers is the constant $d+1$. We can now conclude that a number on diagonal $d+1$ is of the form

$$T_d + c = \frac{d(d+1)}{2} + c = \frac{(r+c-2)(r+c-1)}{2} + c = f(r,c)$$

because on diagonal number $d+1, r+c = d+2 \rightarrow d = r+c-2$. To spot-check that we have figured correctly, note that $f(1,1) = 1, f(2,1) = 2, f(1,2) = 3, f(3,1) = 4$, etc., as expected.

It is interesting (and assuring) to note that unwinding the recurrence in Approach 1 (a very tedious process), yields the same formula for $f(r,c)$. (Note: What we have just derived is known as the *Cantor Pairing Function*).

If the way we arrived at this closed formula does not seem sufficiently rigorous for some, it can be verified by mathematical induction—in this case, *two-dimensional* induction.

Mathematical Induction in Two Variables

To prove by mathematical induction a proposition in two discrete variables, $P(m,n)$, one typically demonstrates the following:

1) $P(1,1)$
2) $P(1,k) \rightarrow P(1,k+1)$
3) $P(h,k) \rightarrow P(h+1,k)$

In other words, we show that the proposition is true for the first (top-left) element, and then that it remains true whenever we move from there horizontally or vertically within the grid of (*m*,*n*) pairs. In our case, $P(r,c)$ is the proposition $n = f(r,c) = \frac{(r+c-1)(r+c-2)}{2}$, where *r* and *c* are the corresponding row and column numbers for the ordinal *n*.

We first show that $P(1,1)$ is true, that is, $f(1,1) = 1$ (the first ordinal). We have $f(1,1) = \frac{(1+1-1)(1+1-2)}{2}+1=0+1=1$. This establishes the base case (formula 1 above).

Now, recall that the elements of the first row are the triangular numbers, which are of the form $\frac{c(c+1)}{2}$, where *c* is the column number. Then, assuming the induction hypothesis

$$f(1,c) = \frac{(1+c-1)(1+c-2)}{2} + c = \frac{c(c-1)}{2} + c = \frac{c(c+1)}{2}$$

we note that this implies $f(1,c+1) = \frac{(c+1)(c+2)}{2}$, which is the next triangular number to the right on the first row, showing that formula 2) above holds.

To establish 3) above, recall that the ordinal in the table in position $(r+1,c)$ is positioned directly below the ordinal in position (r,c). The difference between these two numbers is therefore the length of the diagonal in which (r,c) occurs (which on diagonal *d* obeys the invariant $d = r + c - 1$, as noted earlier). For example, the ordinal 5, which is in position (2,2), is in the 3rd diagonal, and the number directly below it in position (3,2) is 8 = 5 + 3, 3 being the length of the 3rd diagonal. This gives us the relation $f(r+1,c) = f(r,c) + r + c - 1$. We will now establish 3) by showing that $f(r+1,c) = f(r,c) + r + c - 1 = \frac{(r+c)(r+c-1)}{2} + c$, using the inductive hypothesis $f(r,c) = \frac{(r+c-1)(r+c-2)}{2} + c$:

$$f(r,c) + r + c - 1 = \frac{(r+c-1)(r+c-2)}{2} + c + r + c - 1 =$$
$$\frac{(r+c-1)(r+c-2)+2(r+c-1)}{2} + c = \frac{(r+c-1)(r+c-2+2)}{2} + c =$$
$$\frac{(r+c-1)(r+c)}{2} + c \quad \text{QED.}$$

For a discussion on generalized induction, see [4].

Summary

A recent text states, "The single most important skill for a computer scientist is **problem solving**. Problem solving means the ability to formulate problems, think creatively about solutions, and express a solution clearly

and accurately. As it turns out, the process of learning to program is an excellent opportunity to practice problem-solving skills." [5] Our goal as computer science educators is to pass on problem-solving skills using computer algorithms to future generations. The transition from common, procedural problem-solving approaches to effectively deriving recursive relationships and proving them by mathematical induction is a difficult one for most students of computer science. [6] Furthermore, people tend to better retain and master concepts that they take through a *process of discovery and verification* to a complete solution—preferably via working implementations [7]. The examples in this article serve as suitable pedagogical aids in strengthening the mastery of key ways of thinking in the minds of developing computer scientists.

References

1. McMillan,J., "Enhancing College Students' Critical Thinking: A Review of Studies", Research in Higher Education, Vol. 26, No. 1, 1987, p. 3.
2. Calude, Cristian (1994). *Information and randomness. An algorithmic perspective*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag. p. 1.
3. Enderton, H., *Computability Theory: An Introduction to Recursion Theory*, Academic Press, 2011, p. 159–160.
4. Knuth, D., *The Art of Computer Programming*, Volume 1: Fundamental Algorithms, Second Edition, Addison-Wesley, 1973, pp. 20, 468.
5. Downey, Allen B., *Think Python: How to Think Like a Computer Scientist*, O'Reilly, 2012, p. 1.
6. Sooriamurthi, R., "Problems in Comprehending Recursion and Suggested Solutions", ITiCSE, June 2001, pp. 25-28.
7. Bareiss, R., Griss, M., "A Story-Centered, Learn-by-Doing Approach to Software Engineering