

# Four Apt Elementary Examples of Recursion

Edward M. Reingold

Department of Computer Science, Illinois Institute of Technology, 10 West 31st  
Street, Chicago, Illinois 60616-2987, USA. reingold@iit.edu

**Abstract.** We give four elementary examples of recursion that are real-life, non-trivial, more natural than the corresponding iterative approach, and do not involve any sophisticated algorithms, data structures, or mathematical problems. The examples are two forms of writing numbers in words, coalescing page references for an index, and finding unclosed `begin` blocks.

My New Zoo, McGrew Zoo, will make people talk.  
My New Zoo, McGrew Zoo, will make people gawk  
At the strangest odd creatures that ever did walk.  
    I'll get, for my zoo, a new sort-of-a-hen  
    Who roosts in another hen's topknot, and *then*  
    *Another* one roosts in the topknot of his,  
    And another in *his*, and another in HIS,  
And so forth and upward and onward, gee whiz!  
    —Dr. Seuss: *If I Ran the Zoo*

## 1 Introduction

Teaching recursion in elementary programming courses is a critical, but always difficult task. Students have not quite mastered iteration, know no sophisticated algorithms, have not been exposed to advanced data structures, and do not have much feel for the process of turning an inchoate idea of an algorithm into a working program. The most common examples used to illustrate recursion in introductory courses are badly flawed; here are some illustrative examples taken from introductory programming texts:<sup>1</sup>

- Factorials. This is more naturally done by iteration and seems pointless anyway—why would anybody want to compute them? Given integer size limitations, one can only compute a few values anyway.
- Fibonacci numbers. Another seemingly pointless computation. Worse still, the naïve recursive approach leads to exponential computation time.
- Input character/digit reversal. Cute, but of no particular use.
- Towers of Hanoi. An interesting puzzle, but not connected to real life.
- How many ways to make change. A beautiful example of dynamic programming, but who cares how many ways there are to make change for a dollar?

---

<sup>1</sup> For obvious reasons we do not give citations.

When only examples like these are given, the concept of recursion is marginalized. Students come away from such examples with the view (often encouraged by inadequate instructors) that in addition to being difficult to master, recursion is applicable only to puzzles or peculiar mathematical problems, pointless in real life, and likely to lead to inefficient algorithms. This misguided view is exacerbated when instructors introduce important theoretical examples such as Ackermann’s function [1], [2], McCarthy’s “91 function” [3], [4], or Takeuchi’s function [3], [5].

One can, of course, cite many deep, natural, and important uses of recursion, but such examples cannot be presented without substantial algorithmic background or sophisticated data structures. Tree traversals, [6] for instance, are certainly worthy examples of recursive techniques, but to discuss them one needs to have introduced trees; even then, the application of traversals to real problems is beyond the reach of a neophyte programmer. Depth-first search of a graph [7], for another example, is a fundamental technique, enormously important and more natural recursively than iteratively, but it involves a sophisticated data structure (graphs, usually represented as an adjacency structure) and does not seem useful until one has embroidered it in clever ways to determine—in linear time—complex graph properties like connectivity, biconnectivity, triconnectivity, Eulerian paths, and planarity. Recursive descent parsing [8, sec. 4.4], is yet another powerful, beautiful, and useful example of recursion that is too advanced for introductory courses.

In this note we present four examples of recursion that are perfect for elementary programming courses because they involve learning no extraneous material, are difficult to do iteratively but relatively straightforward recursively, and efficiently perform algorithmic tasks of obvious utility. The four examples are two versions of writing numbers in words, coalescing index page references, and finding unclosed `begin` blocks. Our examples are written in a functional style, presented in C-like notation, but without adherence to the syntax of any particular programming language. We assume that arbitrarily large integers can be manipulated.

## 2 Writing Numbers in Words—Naïve Version

We want to write a function that prints positive integers digit by digit, with each digit expressed as a word; that is, for the integer 60203 we want to produce the output “six zero two zero three”. Such a function might be used by the telephone company for an automated information service in which the requested phone number is read digit by digit to the customer. Doing this iteratively is difficult because we need the most significant digit first. Recursively it is simple: We begin with a function that prints (speaks) a single digit:

```
1 printDigit (integer n) {
2 // Write the name of a number n, 0 <= n < 10.
3
```

```

4     case (n) {
5         0: print("zero");
6         1: print("one");
7         2: print("two");
8         3: print("three");
9         4: print("four");
10        5: print("five");
11        6: print("six");
12        7: print("seven");
13        8: print("eight");
14        9: print("nine");
15    else: error("Digit too large");
16    }
17 }

```

where `print` is the basic system print command. Now with integer (truncated) division `div` and modulus `mod` we write

```

1  printDigits (integer n) {
2  // Write n digit by digit in words.
3
4  if (n < 10)
5      printDigit(n);
6  else {
7      printDigits(n div 10);
8      print(" ");
9      printDigit(n mod 10);
10 }
11 }

```

### 3 Writing Numbers in Words—Sophisticated Version

The way we wrote numbers in words in the previous section is inadequate for some purposes. Suppose instead that we want to write a number as it would be spoken; that is, we want the number 8018018851 to be written in words as “eight billion eighteen million eighteen thousand eight hundred fifty one” (Conway and Guy [9, p. 15] call this “Knuth’s number”, the first prime number in the alphabetic ordering of the natural numbers [10, p. 4]). English speakers do this more or less automatically, at least for numbers below one billion, but expressing this algorithmically is subtle. Such an algorithm would be needed for a check-writing program to protect against alteration of the digits in the amount paid. We assume American English nomenclature [11, pp. 12 and 22–24], [12, p. 1549], but the method here is easily adapted to the British English nomenclature, or that of other languages. The imaginative nomenclatures of [9, pp. 14–15] or [13, pp. 311–312] are also easy to accommodate with the ideas presented here.

Our approach, which covers the full range of American English,  $(-10^{66}, 10^{66})$ , is based on [14]; [10, p. 6] has a similar method.

Numbers less than twenty are idiosyncratic, so we handle them with a `case` statement:

```
1  printSmallNumber (integer n) {
2    // Write the name of a number < 20.
3
4    case (n) {
5      1: printString("one");
6      2: printString("two");
7      3: printString("three");
8      4: printString("four");
9      5: printString("five");
10     6: printString("six");
11     7: printString("seven");
12     8: printString("eight");
13     9: printString("nine");
14     10: printString("ten");
15     11: printString("eleven");
16     12: printString("twelve");
17     13: printString("thirteen");
18     14: printString("fourteen");
19     15: printString("fifteen");
20     16: printString("sixteen");
21     17: printString("seventeen");
22     18: printString("eighteen");
23     19: printString("nineteen");
24   else
25     error("Small number too large");
26   }
27 }
```

All output is centralized in the procedure `printString` because it allows us to add spaces, commas, and hyphens between words, as is conventional; we ignore these issues for the moment. We use `printMediumNumber` to handle numbers less than 1000:

```
1  printMediumNumber(integer n) {
2    // Write the name of a number < 1000.
3
4    if (n > 99) {
5      printSmallNumber(n/100);
6      printString("hundred");
7      n = n mod 100;
8    }
9    if (n > 19) {
```

```

10     printDecade(n/10);
11     n = n mod 10;
12 }
13 if (n > 0)
14     printSmallNumber(n);
15 }

```

where the “decade” is written by

```

1  printDecade (integer n) {
2  // Write the name of a multiple of 10.
3
4  case (n) {
5      2: printString("twenty");
6      3: printString("thirty");
7      4: printString("forty");
8      5: printString("fifty");
9      6: printString("sixty");
10     7: printString("seventy");
11     8: printString("eighty");
12     9: printString("ninety");
13 else
14     error("Decade too large");
15 }
16 }

```

Numbers with four or more digits are handled recursively. To express  $n \times 1000^i$  in words, we

express  $\lfloor n/1000 \rfloor \times 1000^{i+1}$  in words,  
express  $n \bmod 1000$  in words, and  
write the name of  $1000^i$  in words.

The last step, writing the name of  $1000^i$ , is done with

```

1  printMillenary (integer n) {
2  // Write the name of the power of a 1000.
3
4  case (n) {
5      1: printString("thousand");
6      2: printString("million");
7      3: printString("billion");
8      4: printString("trillion");
9      5: printString("quadrillion");
10     6: printString("quintillion");
11     7: printString("sextillion");
12     8: printString("septillion");
13     9: printString("octillion");

```

```

14     10: printString("nonillion");
15     11: printString("decillion");
16     12: printString("undecillion");
17     13: printString("duodecillion");
18     14: printString("tredecillion");
19     15: printString("quattuordecillion");
20     16: printString("quindecillion");
21     17: printString("sexdecillion");
22     18: printString("septendecillion");
23     19: printString("octodecillion");
24     20: printString("novemdecillion");
25     21: printString("vigintillion");
26     else
27         error("Millenary too large");
28     }
29 }

```

A “vigintillion” ( $10^{63}$ ) is as high as American nomenclature goes. With `printMillenary` we translate our recursive structure into

```

1  printBigNumber(integer n, integer i) {
2  // Write the name of a number  $n * 1000^i$ ,  $n > 0$ ,  $i \geq 0$ .
3
4      if (n > 0) {
5          printBigNumber(n/1000, i+1);
6          if ((n mod 1000) != 0) {
7              printMediumNumber(n mod 1000);
8              printMillenary(i);
9          }
10     }
11 }

```

Calling `printBigNumber(n, i)` writes  $n \times 1000^i$  in words, so the initial call to `printBigNumber` should have a second parameter of zero. Thus we would write the publicly available code as

```

1  printNumber(integer n) {
2  // Write a number in English words according to
3  // the American nomenclature.
4
5      printBigNumber(n, 0);
6  }

```

with `printString` being simply

```

1  printString (string s) {
2  // Write a string
3

```

```

4   print(s);
5   }

```

However, as the code now stands, a call such as `printNumber(1234)` would produce the output “onethousandtwohundredthirtyfour” because no spaces are introduced between words. To do this properly, we cannot simply write a space after a word (it might be the last word), nor can we simply write a space before each word (it might be the first word). We use a global variable of state (the only acceptable type of global variable!) to tell us whether a space is needed. Similarly, commas and hyphens may be needed in certain contexts. The use of global variables can be avoided by passing the values as parameters to all functions, but because there are many functions here, and several values needed, we illustrate the use of global values as a means of communication across all levels of recursion.

Thus we rewrite `printNumber` as

```

1   printNumber(integer n) {
2   // Write a number in English words according to
3   // the American nomenclature.
4
5   global needBlank = false;
6   printBigNumber(n, 0);
7   }

```

and `printString` as

```

1   printString (string s) {
2   // Write a string, preceded by a blank if needed
3
4   if (needBlank) then print(" ");
5   print(s);
6   needBlank = true;
7   }

```

Ordinarily in English we would write commas after the words “thousand”, “million”, “billion”, and so on. The above code can be adapted to add such commas with a global variable `needComma` that is set to `false` initially in `printNumber` and is set to `true` at the end of `printMillenary`; if the value is `true` in `printString`, a space followed by a comma is printed before the argument is printed, after which the value is reset to `false`. Similarly, some usages require a hyphen between the decade value and the units value (that is, “forty-five”, not “forty five”). Again, we introduce a global variable `needHyphen` that is set to `false` initially in `printNumber`, is set to `true` in `printMediumNumber` after the decade is printed, and is reset to `false` at the end of `printMediumNumber`. If the value is `true` in `printString`, a hyphen (and no space) is printed before the argument is printed.

We can similarly capitalize the first word by introducing a global variable `firstWord`, setting to `true` initially in `printNumber`, modifying `printString`

to capitalize its argument when the value is `true`, and resetting it to `false` as soon as the first word is written. With all of these fillips we can now write

$$2^{219} = 842498333348457493583344221469363-458551160763204392890034487820288,$$

in words as

Eight hundred forty-two vigintillion, four hundred ninety-eight novemdecillion, three hundred thirty-three octodecillion, three hundred forty-eight septendecillion, four hundred fifty-seven sexdecillion, four hundred ninety-three quindecillion, five hundred eighty-three quattuordecillion, three hundred forty-four tredecillion, two hundred twenty-one duodecillion, four hundred sixty-nine undecillion, three hundred sixty-three decillion, four hundred fifty-eight nonillion, five hundred fifty-one octillion, one hundred sixty septillion, seven hundred sixty-three sextillion, two hundred four quintillion, three hundred ninety-two quadrillion, eight hundred ninety trillion, thirty-four billion, four hundred eighty-seven million, eight hundred twenty thousand, two hundred eighty-eight .

Some usage requires the word “and” after the word “hundred”, especially for the rightmost three digits of a number (“one hundred *and* twenty”); this would require another global variable `needAnd` and a slight modification of `printMediumNumber`. Other easy modifications could handle zero and negative numbers properly.

## 4 Indexing Page Numbers

Suppose that we need to convert a sorted sequence of page numbers into an index entry for a book. For example, if a term occurs on pages 1, 3, 4, 6, 7, 8, 9, 12, and 13, the index entry would be

1, 3, 4, 6–9, 12, 13.

We want to write a function that produces this list of pages from the sorted sequence of page numbers. We assume a function `nextPage` that provides the input sequence of page numbers, one by one, in increasing order; the sequence is ended with a negative page number.

To begin, notice that each index entry except the last is followed by a comma. We could use the same technique we used to get blank spaces in the right places when writing numbers in words—a global variable `needComma` that is true when the previous index entry needs to be followed by a comma before we print the next entry. However in this case we illustrate including this value as a boolean parameter. The function to write a single index entry spanning pages `first` to `last` is

```

1  printIndexEntry
2      (integer first, integer last, boolean needComma) {
3  // Print the range of pages first-last as an index entry,
4  // preceded by a comma if needed
5
6      if (needComma)
7          print(", ");
8      if (first == last)          // one-page run
9          print(first);
10     else if (first+1 == last) // two-page run
11         print(first, ", ", last);
12     else                          // run of 3 or more pages
13         print(first, "-", last);
14 }

```

This function is the basis for our scanning a list of pages and printing the index entries; we must call `printIndexEntry` for each run of pages that we find in the list. We scan the list accumulating pages in a single run of pages until the run ends; when the run ends, we call `printIndexEntry`.

As we scan the list, we can represent the current run of pages by its starting and ending page. If the current run stretches from `first` to `last`, then the value of the next page in the list tells us whether to extend the current run or to end it. When the current run ends, we write it with a call to `printIndexEntry` and then, if it was not the final run, use the next page to begin a new run. Thus there are three cases

1. The last run has ended; this happens when there are no more pages in the list, that is, when `nextPage` returns a negative page number.
2. The current run needs to be extended by one page; this happens when the next page in the list is one more than the last page.
3. The current run has ended, but there are more pages in the list; this happens when the next page is *not* one more than the last page and we are not at the end of the list.

Putting this into code we have

```

1  printIndexList
2      (integer first, integer last, boolean needComma) {
3  // Print a list of index entries, preceded by the run
4  // first-last; runs of three or more pages are coalesced
5  // into one hyphenated entry.
6
7      integer page = nextPage();
8      if (page < 0)                          // end of final run
9          printIndexEntry(first, last, needComma);
10     else if (last+1 == page)                 // extend current run
11         printIndexList(first, last+1, needComma);

```

```

12     else {                                     // end current run
13         printIndexEntry(first, last, needComma);
14         printIndexList(page, page, true); // start new run
15     }
16 }

```

Notice that we start a new run (of one page) by making the starting and ending pages equal to the new value given by `nextPage`.

The function as seen by the outside world must initiate `printIndexList` properly with the first index entry becoming the current run. It is here that we must initialize the value of `needComma`; the initial value must be `false` since no comma is needed before we print the first index entry:

```

1  IndexList () {
2  // Print a list of index entries with runs of three or
3  // more pages coalesced into one hyphenated entry.
4
5  integer page = nextPage();
6  printIndexList(page, page, false);
7  }

```

## 5 Finding Unclosed Begins

When editing programs in a block-structured language, one often needs to close the last open block. For example, in  $\text{\LaTeX}$  there are blocks of text that are enclosed in matched, parameterized `begin-end` statements. A  $\text{\LaTeX}$  file might look something like

```

\documentclass{article}

\begin{document}
...
\begin{abstract}
...
\end{abstract}
...
\begin{center}
\begin{itemize}
...
\begin{code}
...
\end{code}
...
\begin{enumerate}
...
\end{enumerate}

```

```

\end{itemize}
\end{center}
...
\begin{quote}
...
\end{quote}
...
\end{document}

```

When editing such a file it is convenient to have an editing command that closes that last unclosed `begin` by adding the appropriate `end` at the current point in the file; for example GNU Emacs [15, sec. 22.9.2] has such a command. But how does the editor find the last unclosed block in a complicated nested arrangement? An easy way is to search backward for `begin` and `end`, keeping a counter whose value starts at 0, is incremented by 1 when we find an `end`, and is decremented by 1 when we find an `begin`. When the counter equals  $-1$  we have found the last unmatched `begin`. However, such a method does not determine if there are mismatched `begin-end` pairs: that is what we want to do.

We assume for simplicity that we have a boolean function `searchBackward` that searches backward from the present location in the file (called the *point*) returning `true` if the search was successful and `false` otherwise, together with a boolean function `lookingAt` that tells us what the text begins at the point. Moreover, we assume that when we call the function that does the search, it changes the point to the start of string found that caused the search to end successfully, but that if the search ends unsuccessfully, the point is unchanged. Finally, we need a string function `match` that returns the shortest non-empty string that matches a substring in a given pattern; that is, `match("end{*}")` returns the shortest string that matches the `*` portion of the argument pattern, starting at the current point.

Here, then, is how we find the last unended `begin`:

```

1  findLastBegin (string s) returns boolean {
2  // Leave point at the beginning of the last unmatched
3  // begin of type s, or of any type if s is empty
4  while (searchBackward("begin" or "end"))
5  do
6  if (lookingAt("begin") && s == "")
7  || lookingAt("begin{ " s "}")) then
8  return true
9  else if (lookingAt("begin")) then
10 return ERROR // improperly matched begin-end pair
11 else // looking at end, so find matching begin
12 return findLastBegin(match("end{*}"))
13 // could not find matching begin of type s
14 return false
15 }

```

The idea, which is quite difficult to express or explain iteratively, is that we search backward from the present location until we find either a `begin` or an `end`. If the search in the `while` condition fails, there is no unmatched `begin` and the function returns `false`. If we find a `begin`, but `s` is empty so that we do not care about the type of `begin`, the search ends successfully—we have found the last unended begin. However if `s` is not empty, we had a specific type of `begin` to find, and if we have found it, the call also ends successfully. But if the search finds a `begin` of a different type, the `begin-end` pairs are mismatched and we signal an error. On the other hand, if the search has found an `end`, we search backward (recursively) to find the matching `begin` to *that end* before continuing in the `while` loop.

## Acknowledgments

The author is grateful to the editors of this volume, Nachum Dershowitz and Ephraim Nissan, and to Benjamin Steele for helpful corrections and suggestions.

## References

1. Ackermann, W.: Zum hilbertschen aufbau der reellen zahlen. *Mathematische Annalen* **99**(1) (1928) 118–133
2. Wichmann, B.A.: Ackermann’s function in Ada. *Ada Lett.* **VI**(3) (1986) 65–70
3. Knuth, D.E.: Textbook examples of recursion. In: *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*. Academic Press Professional, Inc., San Diego, CA, USA (1991) 207–229
4. Manna, Z.: *Mathematical Theory of Computation*. McGraw-Hill, New York, NY, USA (1974)
5. Takeuchi, I.: On a recursive function that does almost recursion only. Technical report, Musahino Electrical Communication Laboratory, Nippon Telephone and Telegraph Company, Tokyo, Japan (1978)
6. Knuth, D.E.: *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd edn. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1997)
7. Tarjan, R.E.: Depth-first search and linear graph algorithms. *SIAM Journal on Computing* **1**(2) (1972) 146–160
8. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers—Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., Inc., Reading, MA, USA (1986)
9. Conway, J.H., Guy, R.: *The Book of Numbers*. Springer-Verlag, New York, NY, USA (1996)
10. Knuth, D.E., Miller, A.A.: A programming and problem-solving seminar. Technical Report STAN-CS-81-863, Department of Computer Science, Stanford University, Stanford, CA, USA (June 1981)
11. Davis, P.J.: *The Lore of Large Numbers*. Yale University Press, New Haven, CT, USA (1961)
12. Gove, P.B.: *Webster’s Third New International Dictionary of the English Language*. G. & C. Merriam Co., Springfield, MA, USA (1961)

13. Knuth, D.E.: Supernatural numbers. In Klärner, D.A., ed.: The Mathematical Gardner. Wadsworth, Boston, MA, USA (1981) 310–325
14. Reingold, E.M.: Writing numbers in words in T<sub>E</sub>X. TUGboat **28**(2) (2007) 256–259
15. Stallman, R.: GNU Emacs Manual. 16th edn. Free Software Foundation, Boston, MA, USA (2007)