23

Converting Numbers into English Words

Lothar Schmitz

Universität der Bundeswehr München, Munich, Germany

The problem we are considering here is how to convert numbers into English words. This is what we naturally do when talking to somebody or when filling out a cheque. An amount of, say, $ 31,264 would be pronounced as "thirty-one thousand two hundred and sixty-four dollars." In contrast, telephone numbers are often articulated digit by digit. So tel. 31264 would be pronounced as "telephone three-one-two-six-four."

To indicate a distance of 1,723 miles to New York, the software of a modern GPS route guidance system would not use telephone style

one-seven-two-three miles to New York

Rather, we expect its friendly voice to say (like we do)

one thousand seven hundred and twenty-three miles to New York

A route guidance system for outer space would have to be able to pronounce very large numbers. For example, 12,345,678,987,654,321 would have to be spoken as

twelve quadrillion
three hundred and forty-five trillion
six hundred and seventy-eight billion
nine hundred and eighty-seven million
six hundred and fifty-four thousand
three hundred and twenty-one

In principle, we all can do that (given the names of very large cardinals). But how would a computer program solve this problem? Here, the details will probably turn out to be tricky! We start by precisely specifying the problem to be solved.

Problem: Given a natural number x satisfying $1 \leq x < 10^{27}$ generate the English wording for x. For simplicity, we assume that numbers of this size can be represented in the programming language we are using and that comparison

B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_23,

operators as well as basic arithmetic operations like addition, subtraction, multiplication and integer division are available for these numbers.

Stepwise Development of an Algorithm

The simplest method would be to store for each number its associated English wording and to retrieve the wordings on demand. Because of its enormous storage requirements this approach is not viable. Note that memorizing such an amount of data would overstrain our human brains as well. Instead, we systematically generate number wordings each time we need them. Obviously, this allows us to keep far less data in our brains.

Now let us try to become aware of and explicitly describe the method we are unconsciously applying ourselves when generating number wordings! How does that method work?

Usually, large numbers are separated by commas into groups of three digits, starting with the three last digits – those with the lowest weight. This indicates how our mental processing works: We first split large numerals into groups of three digits each. For the last example this gives:

ones: 321
thousands: 654
millions: 987
billions: 678
trillions: 345
quadrillions: 12

We observe that each group denotes an integer between 0 and 999. The leftmost group may have fewer than three digits (but at least one digit).

Now the groups are processed from left to right. For each group first the wording of its associated number (an integer between 0 and 999) is generated. To this an indication of the group's weight is attached: "quadrillions," "trillions," "billions," etc. A weight indication for ones is simply left out.

When generating the wording of a number between 0 and 999 we process the digits from left to right, i.e., in decreasing weight order: hundreds, tens, and ones. These weights independently may occur or not. Hundreds are joined to the following digits (if any) with an "and." Likewise, tens and following ones are joined with a hyphen. There are a number of special cases like "twelve" and "seventeen", where tens and ones are merged into one word. More subtleties will turn up later in the program development.

Splitting Numbers into Three-Digit Groups ...

In order to split a number into three-digit groups, we repeatedly divide the number by 1000 and thus each time obtain the next three-digit group. Step 1

below additionally computes the number i of three-digit groups contained in the given *number*.

STEP 1: Splits number into three-digit groups. The three-digit groups are stored in array *group*: the group with weight 1 in *group*[0], the group with weight 1,000 in *group*[1], the group with weight 1,000,000 in *group*[2], and so on. Variable i records the number of three-digit groups found so far.

```
1  i := 0      // initially no group found
2  while number ≥ 0 do
3      group[i] := number mod 1000      // remainder and ...
4      number := number/1000      // quotient when dividing by 1000
5      i := i + 1      // one more group found
6  endwhile
```

... and Generating the English Words

In STEP 2 below, the more complex parts are deferred to the auxiliary functions GENERATEGROUP and GENERATEWEIGHT, which compute the English wording of a three-digit group and its weight, respectively.

STEP 2: Generating the English words. Indices in array *group* range from 0 to $i-1$, for the variable i computed in STEP 1. The index of the leftmost group (i.e. the one to be translated first) is $i-1$. Variable *text* records the English number text generated so far. The & operator joins ("concatenates") two pieces of text into one piece.

```
1  text := ""      // initially text is empty
2  i := i − 1      // index of the leftmost group
3  while i ≥ 0 do
4      text := text & GENERATEGROUP(group[i])      // generate words ...
5      text := text & GENERATEWEIGHT(i)
           // ... for group and its weight
6      i := i − 1      // on to the next group
7  endwhile
```

After executing both STEP 1 and STEP 2 in order, variable *text* contains the English wording of the given number, as required!

Function generateGroup

If a three-digit group denotes the value 0, it does not contribute to the English wording of the number. This is demonstrated by an example: The number 1,000,111 is spoken "one million one hundred and eleven." The middle group 000 denoting the value 0 indeed does not show in the generated text.

Three-digit groups denoting values greater than 0 are split into three digits: digit h with weight 100, digit t with weight 10, and o with weight 1.

Here, we cannot proceed without knowing the English names of the Arabic digits and a few other names that are used to systematically build up the

English numbers. Since numbers less than 20 are not built as regularly as numbers beyond 19, we simply store the full names of numbers less than 20 in an array *lessThan*20 of strings. Likewise, we store the names of multiples of 10 (between 20 and 90) in an array *times*10 of strings. This results in:

Declaration of arrays with small numbers (between 1 and 19) and multiples of 10. For i between 1 and 19 we find the name of i in *lessThan*20[i]. For j between 2 and 9 the element *times*10[j] contains the name of $j \cdot 10$.

```
1  lessThan20: array [0..19] of string :=
2      [ "", "one", "two", "three", "four", ..., "eighteen", "nineteen" ]
3  times10: array [2..9] of string :=
4      [ "twenty", "thirty", "forty", ..., "eighty", "ninety" ]
```

We are now prepared to define the core function GENERATEGROUP, which generates the English wording of a number between 0 and 999. If (part of) a number has value 0, its generated text will be empty. The text will be built up in the string variable *words*, which initially is empty. First, digit h is translated, then the rest r of the number. If $r < 20$, the translation is taken from array *lessThan*20[r]. Otherwise, r is split into the two digits, t and o, which are translated into words using the arrays *times*10 and *lessThan*20, respectively. Non-empty hundreds and rests are joined with the word "and". Likewise, non-empty translations of t and o are joined with a hyphen. Keep in mind, that practically all number components may be missing. Note how spaces separating words are inserted into the text.

Translate a *number* between 0 and 999 into English text.

```
1  function GENERATEGROUP(number)
2      h := number/100
3      r := number mod 100
4      t := r/10
5      o := r mod 10
6      words := ""
7  begin
8      if h > 0 then words := words & lessThan20[h] & "hundred " endif
9      if h > 0 and r > 0 then words := words & "and " endif
10     if r < 20 then
           // for r = 0 this works because of lessThan20[0] = ""
11         words := words & lessThan20[r]
12     else
13         if t > 0 then words := words & times10[t]
14         if t > 0 and o > 0 then words := words & "-"
15         if o > 0 then words := words & lessThan20[o]
16     endif
17     return words
18 end
```

Function generateWeight

The English names of the weights, like the names of Arabic digits, must simply be known. For this purpose we introduce an array *weight* of strings. Recall that a weight of 1 is "not spoken."

Declaration of array *weight*.

```
1   weight : array [1..8] of string :=
2       [ "", "thousand", "million", "billion", "trillion",
3         "quadrillion", "quintillion", "sextillion", "septillion" ]
```

Compared to other languages, English allows the text of numbers to be generated very systematically, almost without grammatical irregularities. One exception is that if the last part is less than one hundred it is joined to the preceding text with an "and" even if there are no preceding hundreds. For example, 4,000,001 is pronounced as "four million *and* one" and 5,004,003 as "five million four thousand *and* three."

Function GENERATEWEIGHT below takes all this into account and also inserts blanks to properly separate the wordings of weights and three-digit groups from each other. If you want to change any of these features, function GENERATEWEIGHT is the place for modifications.

Generate the weight wording for the ith group. Ensure that all words are separated by blanks.

```
1   function GENERATEWEIGHT(i)
2       words := ""
3   begin
4       if group[i] > 0 then
5           words := " "  & weight[i] &  " "
6       endif
7       if i = 1 and group[0] < 100 and group[0] > 0 then
8           words := words &  "and"
9       endif
10      return words
11  end
```

Lessons Learned

We now understand how to program the speech output for a GPS route guidance system. Since we have tried to model our algorithm using our own, intuitive approach we now also better understand the way humans generate English number texts.

It is surprising how little we have to know "by heart" in order to be able to generate a multitude of English numbers. All primitive names "known to" the algorithm are stored in the arrays *lessThan*20, *times*10, and *weight*. A total of 36 short strings suffices for generating almost 10^{27} English number names – more than we can use in our whole life (100 years have only about 3,153,600,000 seconds)!

And it goes on like this: In order to increase the range of numbers that can be generated by a factor of 1,000 you need to extend the *weight* array by only one more string entry. You can continue this way as long as you find correct weight names (many are listed on the web site http://home.hetnet.nl/~vanadovv/BignumbyN.html).

For other languages, say French or German, that also use a positional number system, number names can be generated "in principle" in the same way. As a consequence, we probably can adapt our algorithm to these languages rather easily. Actually, the original version of our algorithm was written for German numbers. More about that below!

Adaptability to changing requirements is an important property of modern software – for our algorithm obvious modifications would be to extend the number range or to produce numbers in other natural languages as indicated above. Often, this is accomplished by storing data that will probably be changed in some data structure. For our algorithm we have achieved adaptability by storing all name primitives in the three arrays *lessThan*20, *times*10, and *weight*.

What to Read and Try out for Yourself

1. The Wiktionary page http://en.wiktionary.org/wiki/Appendix:Numbers

   Here, you can learn interesting facts about number representation systems, e.g., what names there are for very large numbers. Among other things, we learn that millions and billions are followed by trillions, quadrillions and quintillions, and that a "googol" corresponds to the number 10^{100}, that is, a 1 followed by 100 zeroes!

2. Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

   In Sect. 5.1 of this textbook, a Haskell program is developed in detail that allows you to generate English numbers smaller than one million.

3. Write a "real program."

   If you try to code the algorithm in your favorite programming language, in order to obtain an executable program, you will probably find very soon that the finite representations typically used for natural numbers (`int`, `integer`, `long` or something similar) are not suitable for storing the big numbers we are discussing here. Actually, it is much better to store decimal

constants like 12345678987654321 as strings (i.e., "12345678987654321") in the first place. In the algorithm, arithmetic operations like *number* **mod** 1000 and *number*/1000 are only used to access and delete the last three-digit group, respectively.

4. Processing even larger numbers.

   As already indicated this algorithm can easily be adapted to handle larger numbers. You only have to extend the array *weight* accordingly. If you want your program to be able to handle novemdecillions (10^{60}), you have to add a dozen strings. Other simple extensions are to include zero and negative numbers.

5. Speaking other languages.

   It takes more effort to adapt the program to another natural language: Obviously, all the primitive names (i.e., the contents of the three arrays *lessThan*20, *times*10, and *weight*) have to be replaced. The trickier part is to adapt the functions GENERATEGROUP and GENERATEWEIGHT to the grammatical peculiarities of the new target language. For example, in German numbers the order of the last two digits of every three-digit group is inverted, like in "one-and-twenty." Also, the different grammatical genders of words and different singular and plural forms require different versions of the *weight* array to be used. If you buy the German original of this book (*Taschenbuch der Algorithmen*, Springer, 2008), you can find out all the details.