

A Solution to the *Word Numbers* Puzzle

Alan Saqui
alan.saqui@gmail.com

July 28, 2008

Abstract

Word Numbers is a puzzle given on ITA Software’s website, as part of their recruitment program for software engineers. In this paper, I give a complete description of the *Word Numbers* puzzle, and sketch out my method for solving it with a reasonable amount of computation. However, in the interest of fairness I have omitted the actual answer to the puzzle as well as my implementation of the solution from this paper.

1 Introduction

1.1 Problem Description

The *Word Numbers* puzzle is one in a series of programming puzzles proposed by ITA Software as part of their recruitment program for new software engineers. A complete description of the puzzle is as follows:

“If the integers from 1 to 999,999,999 are written as words, sorted alphabetically, and concatenated, what is the 51 billionth letter?”

To be precise: if the integers from 1 to 999,999,999 are expressed in words (omitting spaces, ‘and’, and punctuation¹), and sorted alphabetically so that the first six integers are

- eight
- eighteen
- eighteenmillion
- eighteenmillioneight
- eighteenmillioneighteen
- eighteenmillioneighteenthousand

and the last is

¹For example, 911,610,034 is written “ninehundredelevenmillionsixhundredtenthousandthirtyfour”; 500,000,000 is written “fivehundredmillion”; 1,709 is written “onethousandsevenhundrednine”.

- twothousandtwohundredtwo

then reading top to bottom, left to right, the 28th letter completes the spelling of the integer “eighteenmillion”.

The 51 *billionth* letter also completes the spelling of an integer. Which one, and what is the sum of all the integers to that point?

This description is available on ITA Software’s website (<http://www.itasoftware.com/careers/SolveThisWorkHerePuzzles.html>).

1.2 Brute-Force Attempt

At first glance, this puzzle seems solvable via ‘brute-force’: that is, to simply write out all of the numbers from 1 to 999,999,999 as words, lexicographically sort them, and then go down the list, taking the summation of the numbers’ values as well as their lengths until you reach 51,000,000,000. However, some quick calculations show that this requires a great deal of space and computing power.

Writing out the numbers 1 to 999,999,999 in words would require over 71,000,000,000 characters, including null terminators (how we arrived at this number will be shown later in the paper). Assuming a character is 1 byte, this would require roughly 71 GB of memory . To store the numbers themselves, if we assume an integer is 4 bytes long then this would require an additional 4 GB of memory. Finally, given an ideal sorting algorithm with $O(n \log n)$ running time, the system would need to perform at least 9 billion computations. Although these numbers are not intractable by today’s standards, it would be desirable to find a solution that is efficient enough to run on commodity hardware.

1.3 Solution Overview

My solution to this puzzle is composed of two main ideas:

1. Being able to easily convert a number into its word representation as given in the puzzle description.
2. Being able to efficiently store a lexicographically sorted list of the numbers 1 to 999,999,999.

The word representation of any number can be decomposed into 3 parts: the ‘hundreds’ section, the ‘thousands’ section and the ‘millions’ section. Each of these sections follows the same format: the word representation of a number from 0 to 999, followed by a suffix: ‘million’ for the ‘millions’ section, ‘thousand’ for the ‘thousands’ section, and ‘’ (no suffix) for the ‘hundreds’ section. For example, the number 17,400,369 can be decomposed into 17, 400, and 369 which then become ‘seventeenmillion’, ‘fourhundredthousand’, and ‘threehundredsixtynine’ respectively.

When storing the numbers in lexicographical order, an important observation is that numbers starting with a given ‘thousands’ or ‘millions’ prefix

will be grouped together in the the list. For example, all numbers starting with ‘**eighteenmillion**’ will be grouped together, as will all numbers starting with ‘**eighteenthousand**’. The lexicographical ordering also ensures that these groupings have very similar structures. For example, the grouping for ‘**eighteenthousand**’ (as numbers) is:

18000, 18008, 18018, ... 18202

Similarly, the grouping for ‘**fiftytwothousand**’ (as numbers) is:

52000, 52008, 52018, ... 52202

The structure of these groupings allows us to calculate ahead of time the total number of characters in a grouping, as well as the sum of all the numbers in a grouping. This greatly improves the efficiency of scanning to the 51 billionth letter because now it is possible to ‘skip over’ entire groups of numbers at a time.

The following sections provide implementation details for this approach.

2 Transforming Numbers into Words

2.1 Creating the Lookup Table

The NumToWord algorithm takes a number a (where $1 \leq a \leq 999999999$) as input and returns its word representation. To create this algorithm, the first step is to build a lookup table T_{words} that maps the numbers 1 to 999 to their word representation.

T_{words} is first initialized for the numbers 0 to 19 (with 0 mapping to the empty string) and all multiples of 10 less than 100.

$$T_{words} = \begin{pmatrix} 0 & \rightarrow & '' \\ 1 & \rightarrow & \text{one} \\ 2 & \rightarrow & \text{two} \\ \vdots & \vdots & \vdots \\ 19 & \rightarrow & \text{nineteen} \\ 20 & \rightarrow & \text{twenty} \\ 30 & \rightarrow & \text{thirty} \\ \vdots & \vdots & \vdots \\ 90 & \rightarrow & \text{ninety} \end{pmatrix}$$

The rest of the T_{words} can now be filled in (Algorithm 1). This is done in two parts: first, the two-digit numbers are filled in by doing a lookup in T_{words} on the value of the tens digit, doing a lookup in T_{words} on the ones digits, and then concatenating the results. For example, for the number 42, $T_{words}[40] = \text{‘forty’}$ and $T_{words}[2] = \text{‘two’}$, so $T_{words}[42] = \text{‘fortytwo’}$.

Next, the three-digit numbers are filled in by doing a lookup in T_{words} on the hundreds digit (not its actual value), doing a lookup in T_{words} on the rest

of the digits, then concatenating the results with the suffix ‘hundred’. For example, for the number 694, $T_{words}[6] = \text{‘six’}$, $T_{words}[94] = \text{‘ninteyfour’}$, so $T_{words}[694] = \text{‘sixhundredninteyfour’}$.

```

input :  $T_{words}$ 
output:  $T_{words}$  filled in for values 1 to 999
for  $a = 21$  to  $999$  do
  if  $21 \leq a < 100$  then
     $T_{words}[a] := T_{words}[[a \div 10] \cdot 10] + T_{words}[a \bmod 10]$ ;
  else
     $T_{words}[a] := T_{words}[[a \div 100]] + \text{‘hundred’} + T_{words}[a \bmod 100]$ ;
  end
end

```

Algorithm 1: Algorithm to fill in T_{words}

2.2 The NumToWord Algorithm

With T_{words} completely filled in, a full description of the NumToWord algorithm can now be given (Algorithm 2). For numbers less than 1000, a lookup in T_{words} is performed. For numbers greater than or equal to 1000, the number is first ‘decomposed’ into its ‘hundreds’, ‘thousands’, and ‘millions’ sections of digits. A lookup in T_{words} is performed on each section. The appropriate suffix is appended, and the results concatenated together. Note that NumToWord calls itself when dealing with a number larger than 999999. This is to properly handle cases where the number has no ‘thousands’ group, such as 437,000,918.

```

input : An integer  $a$ , where  $1 \leq a \leq 999,999,999$ 
output: The word representation of  $a$ 
if  $1 < a < 999$  then
  return  $T_{words}[a]$ 
else if  $1,000 \leq a \leq 999,999$  then
  return  $T_{words}[[a \div 1000]] + \text{‘thousand’} + T_{words}[a \bmod 1000]$ 
else
  return  $T_{words}[[a \div 1,000,000]] + \text{‘million’} +$ 
   $\text{NumToWord}(a \bmod 1,000,000)$ 
end

```

Algorithm 2: NumToWord

3 Sorting A Billion Numbers Lexicographically

3.1 Building Look-aside Lists

The first step in creating the sorted list of numbers is to lexicographically sort the numbers 1 to 999 using the NumToWord algorithm. Call this list the 'hundreds' list, or L_h :

$$L_h = 8, 18, 808, 800, 818, 880, 888, \dots, 222, 202$$

L_h can be expanded to include the numbers from 1000 to 9999 via the following observation: let a be some multiple of 1000. Then, for all numbers $a, a + 1, a + 2, \dots, a + 999$, these numbers must be grouped together in the sorted list, since they all start with NumToWord(a). For example, if $a = 1000$, then the numbers 1000 to 1999 all start with the phrase 'onethousand', and so must be grouped together in the sorted list.

The order of each of these groups can be determined by decomposing the word representation of a number n in the group into a 'prefix' of NumToWord($n - (n \bmod 1000)$) and a 'suffix' of NumToWord($n \bmod 1000$). For example, the group for 1000 is as follows:

$$G_{1000} = \begin{array}{rcll} 1000 & \rightarrow & \text{'onethousand'} & + & '' \\ 1008 & \rightarrow & \text{'onethousand'} & + & \text{'eight'} \\ 1018 & \rightarrow & \text{'onethousand'} & + & \text{'eighteen'} \\ 1808 & \rightarrow & \text{'onethousand'} & + & \text{'eighthundredeight'} \\ \vdots & \rightarrow & \vdots & + & \vdots \\ 1222 & \rightarrow & \text{'onethousand'} & + & \text{'twohundredtwentytwo'} \\ 1202 & \rightarrow & \text{'onethousand'} & + & \text{'twohundredtwo'} \end{array}$$

Note that the last column in the group construction is simply L_h . Therefore, the entire list can be reconstructed by appending the prefix to every element in L_h .

L_h can now be augmented with the numbers from 1000 to 999999 by adding the multiples of 1000 less than 1000000 and resorting the list. Call this list the 'hundreds and thousands' list, or L_{ht} :

$$L_{ht} = 8, 18, 18000, 800, 808, 818, 818000, \dots, 222, 222000, 202, 202000$$

Finally, L_{ht} can be expanded to include the numbers from 1,000,000 to 999,999,999 via a similar method. Let a be some multiple of 1,000,000. Then, for all numbers $a, a + 1, a + 2, \dots, a + 999999$, these numbers must be grouped together in the sorted list, since they all start with NumToWord(a). For example, if $a = 1,000,000$, then the numbers 1,000,000 to 1,999,999 all start with the phrase 'onemillion', and so must be grouped together in the sorted list.

The order of each of these groups can be determined by decomposing the word representation of a number n in the group into a 'prefix' of NumToWord($n - (n \bmod 1,000,000)$) and a suffix of NumToWord($n \bmod 1,000,000$). For example, the group for 1,000,000 is as follows:

| | | | | | |
|--------------|-----------|---|--------------|---|-------------------------|
| $G_{1000} =$ | 1,000,000 | → | ‘onemillion’ | + | ‘ |
| | 1,000,008 | → | ‘onemillion’ | + | ‘eight’ |
| | 1,000,018 | → | ‘onemillion’ | + | ‘eighteen’ |
| | 1,018,000 | → | ‘onemillion’ | + | ‘eighteenthousand’ |
| | 1,000,800 | → | ‘onemillion’ | + | ‘eighthundred’ |
| | ⋮ | → | ⋮ | + | ⋮ |
| | 1,000,202 | → | ‘onemillion’ | + | ‘twohundredtwo’ |
| | 1,202,000 | → | ‘onemillion’ | + | ‘twohundredtwothousand’ |
| | 1,002,000 | → | ‘onemillion’ | + | ‘twothousand’ |

Note that the last column is simply L_{ht} . Therefore, the entire list can be reconstructed by appending the prefix to every element in L_{ht} .

L_{ht} can now be augmented with the numbers from 1,000,000 to 999,999,999 by adding the multiples of 1,000,000 less than 1,000,000,000 and resorting the list. Call this list the ‘hundreds, thousands and millions’ table, or L_{htm} :

$$L_{htm} = 8, 18, 18000000, 18000, \dots, 202000, 2000000, 2000$$

L_{htm} now represents a lexicographically sorted list of all the numbers from 1 to 999,999,999.

3.2 Calculating List and Group Sizes

The WordNumbers puzzle calls for calculating two numbers when scanning through the list of numbers: the total number of letters seen thus far, and the summation of all numbers seen thus far. Therefore, since the lists constructed in the previous section contain groups of numbers, a method to calculate the total number of letters in each group, as well as the sum of all numbers in each group is required.

The sum of all numbers in each group can be calculated by using the following summation formula:

$$\sum_{i=m}^n i = \frac{(n-m+1)(n+m)}{2}$$

For a ‘thousands’ group a , $m = a$ and $n = m + 999$, since each thousands group contains the numbers $a, a + 1, a + 2, \dots, a + 999$. For a ‘millions’ group a , $m = a$ and $n = m + 999999$ since each millions group contains the numbers $a, a + 1, a + 2, \dots, a + 999999$.

Let $\text{len}()$ be a function that, given a word representation of a number, returns its length (or the total number of characters in the given word representation). Then, the length of L_h can be calculated by applying $\text{len}()$ to every element in L_h and adding the results.

Now that we have the total number of characters in L_h , the length of any thousands group can be calculated. As shown in the previous section, the numbers in a thousands group can be decomposed into a prefix applied to every

element in L_h . Each thousands group consists of 1000 such numbers. Therefore, there are 1000 prefixes in the group. This gives the following total for any thousands group a :

$$\text{length of group } a = 1000 * \text{len}(\text{NumToWord}(a)) + \text{length of } L_h$$

The length of L_{ht} can be obtained by applying this formula to every multiple of 1000 in L_{ht} , and adding the results to the length of L_h .

The length of any millions group is obtained by expanding this method. Again, the previous section showed that the numbers in a given millions group can be decomposed into a prefix applied to every element in L_{ht} . Each group consists of 1,000,000 such numbers. Therefore, there are 1,000,000 prefixes in the group. This give the following total for any millions group a :

$$\text{length of group } a = 1,000,000 * \text{len}(\text{NumToWord}(a)) + \text{length of } L_{ht}$$

Continuing this method to calculate the total length of L_{htm} gives the character total mentioned in the introduction.

3.3 Scanning the List

The **FindBreaker** algorithm and **FindExactBreaker** algorithm work by scanning the list L_{htm} , looking for a ‘breaker’, or a number that, when added to the running total, puts len at or over the limit. The algorithms skip over the thousands and millions groups by using the calculations from the previous section. If the breaker is itself a group, then the algorithms ‘drill down’ into the group by expanding it to find the exact number in the group that is the breaker.

FindBreaker is shown in Algorithm 3. If the number num is a multiple of 1,000 or 1,000,000, the algorithm will take num as being a marker for a group of numbers and use the sum and length of the entire group. When **FindBreaker** reaches a number that pushes len over the character limit, it calls **FindExactBreaker** to drill down into the breaker and find the exact number within the group that is the breaker.

FindExactBreaker is shown in Algorithm 4. It first checks the breaker itself (not the group it represents) to see if it pushes len over the limit, and returns the breaker if it does. If the breaker itself doesn’t push len over the limit, then the length of the breaker is added to len .

FindExactBreaker then examines the breaker to see if it is a number that represents a group. If the number represents a millions group, then the algorithm uses L_{ht} to expand the group. If the number represents a thousands group, then the algorithm uses L_h instead.

Once the proper list is loaded, **FindExactBreaker** then scans the list in the same manner as **FindBreaker**, looking for a number that pushes len over the limit. Once a breaker is found, it is sent through the algorithm again, since the new breaker could itself be a group. The final number that is returned is the exact breaker.

```

output: The number  $a$  that puts the total number of characters seen at
or over limit
limit  $\leftarrow$  51,000,000,000 ;
sum  $\leftarrow$  0 ;
len  $\leftarrow$  0 ;
foreach  $num$  in  $L_{htm}$  do
    numlen  $\leftarrow$  length of  $num$  ;
    if len + numlen  $\leq$  limit then
        breaker  $\leftarrow$   $num$  ;
        break ;
    end
    sum  $\leftarrow$  sum + sum of  $num$  ;
    len  $\leftarrow$  len + length of  $sum$  ;
end
return FindExactBreaker(sum, len, breaker)

```

Algorithm 3: FindBreaker

```

/* Length of breaker itself, not the table it represents */
if actual length of breaker + sum > limit then
    return breaker;
else
    /* Sum and length of breaker, not the table it represents */
    sum  $\leftarrow$  sum + actual sum of breaker;
    len  $\leftarrow$  len + actual length of breaker;
end
if breaker mod 1000000 = 0 then /* millions group */
    list =  $L_{ht}$ ;
else if breaker mod 1000 = 0 then /* thousands group */
    list =  $L_h$ ;
else
    return breaker;
end
foreach  $num$  in list do
     $num$   $\leftarrow$   $num$  + breaker;
    if len + length of  $num$   $\leq$  limit then
        breaker  $\leftarrow$   $num$ ;
        break;
    end
    sum  $\leftarrow$  sum + sum of breaker;
    len  $\leftarrow$  len + length of breaker;
end
return FindExactBreaker(breaker)

```

Algorithm 4: FindExactBreaker

4 Conclusions

4.1 Implementation

Because of the size of the numbers involved in this puzzle, it is necessary to use a programming language with arbitrary-sized integers. My first attempt at implementing this solution was with perl, using the `Math::BigInt` module, but my final implementation was in python. I ran it on a Pentium III 1.5 GHz machine running Windows XP and Python v. 2.5. The program finished within a few seconds, having examined roughly 3500 numbers and groups before finding the exact breaker. This is a tremendous speed-up from the brute-force method described in the introduction. The implementation itself is about 200 lines long.

4.2 Further Work

The solution outlined in this paper could easily be expanded to larger ranges of numbers, say up to 999,999,999,999. The work would involve creating a new table, L_{htmb} , that contained markers for the billions groups, and an alteration of the `NumToWord`, `FindBreaker`, and `FindExactBreaker` algorithms by adding cases for this new type of group. L_{htmb} would contain only an additional 1,000 entries, so the overall performance of the algorithm shouldn't be impacted too much.

5 Acknowledgments

I would like to thank Adam Calabrese for encouraging me to work on the *Word-Numbers* puzzle, as well as Alex Gordon for providing me with some good discussions. Finally, I'd like to thank ITA Software for providing this puzzle.