Software Explorations

THE BIG SQUEEZE
BY JON BENTLEY

English text has a lot of redundancy. People are pretty good at squeezing out redundancy and then expanding it back again. That's why we can read advertisements like this:

> WASH DC Lg furn hse avail Jan 93. Move-in cond! 3 stories, form din rm (almost nu china incld), ball rm, fam rm, fin bsmt w/sep entr, wd flrs, fpls, lg encl yrd w/vu, xcel loc nr pk, bulletprf wndws, w/many charm architect details.

Data compression is important for computers, too. Squeezing data can reduce the size of disk files, cut telephone transmission costs, and increase the effective bandwidth of video channels. In this column we'll study compression techniques by tackling a venerable problem.

The Problem

Let's start with the rules of the game. Our problem is to see how much we can squeeze a dictionary. We don't need to implement compression and expansion programs. Our task is only to measure the effect of the compression — prototyping should always precede building. We can use any widely available UNIX tool.

Our dictionary is really a sorted word list: it contains 71661 words, one word per line, for a total of 691752 characters. There is no additional information, such as pronunciation, definition, or etymology. All characters are lower-case letters. We'll soon see the first few lines of the dictionary. I made the dictionary by translating all capital letters in my system's dictionary to lower case, deleting words that contain nonalphabetic characters, and then removing duplicate words:

```
tr 'A-Z' 'a-z' </usr/dict/words |
  grep -v '[^a-z]' | uniq >dict.in
```

> **Exercise 1.** Before you read further, jot down some methods that you might use to compress the dictionary, and estimate the effectiveness of each. All-in-all, by what factor do you think you could squeeze the dictionary?

The clean and precise rules identify this task as a classroom exercise, but similar problems arise in a host of applications. A list of English words is a critical component in programs like text editors and spelling checkers. A compressed dictionary takes less space to store,

less time to transmit over communication lines, and is often faster to read from disk.

Quick and a Fair Code, Five Bits

Each character in the dictionary is represented on my machine by an 8-bit byte. But 8 bits can represent $2^8 = 256$ different characters, and we are wasting them on just 27 different characters (26 letters and newline). Because $2^5 = 32$, we can represent our dictionary using a 5-bit code. This will reduce the space required by the dictionary to just 62.5% of the original representation.

According to the rules of our game, we've solved the problem. We have described a scheme and analyzed its performance. We could leave the encoding and decoding programs as ''implementation details'' that are outside the scope of this exercise. (The best programmers I know have big lazy streaks right down the middle of their backs.)

Exercise 2. Implement programs to encode and decode 5-bit characters.

But for consistency with later programs (and for use in later pipelines), I wanted to have a program that would tell how many bits a file uses under a five-bit encoding in a command like this:

```
bitcost 5 dict.in
```

Here is a shell implementation of `bitcost`:

```
expr $1 "*" `wc -c $2`
```

The `-c` tells word count to count the number of characters; the backquotes cause the output to be placed in the command line. The `expr` program evaluates its arguments as an arithmetic expression.

Exercise 3. Write a `bitcost` program that reads its standard input if it is not given a second argument. Try implementing `bitcost` using other tools, such as Awk or `ls`. Extend `bitcost` to check whether there are at most 2^{bits} distinct characters in the file.

Common Prefix Elimination

A common technique in data compression is ''differential coding''. If we are transmitting a series of similar objects, we send only the differences between objects. In television images, for instance, where large chunks of background are unchanged we may transmit only the changes from the last screen image.

Because the words in the dictionary are sorted, a typical word has many letters in common with its predecessor. We'll therefore represent each word by a count of the letters it has in common with its predecessor followed by the letters that differ. The original dictionary is shown on the left, and the encoded version is on the right:

```
a           0a
aardvark    1ardvark
aardwolf    4wolf
aaron       3on
aaronic     5ic
ab          1b
aba         2a
abaca       3ca
abaci       4i
aback       4k
   ...
```

The third line asserts that the word ‘‘aardwolf’’ has 4 letters in common with ‘‘aardvark’’, and then the letters ‘‘wolf’’.

Here is an Awk implementation of the `preflen` program to perform prefix-length encoding:

```
{ for (i = 1; substr($1, i, 1) ==
          substr(last, i, 1); i++) ;
  print i-1 substr($1, i)
  last = $1
}
```

Because there is no pattern, the action in braces is executed for each input line. The `for` loop in the first two lines sets `i` to the number of characters that the input word has in common with the `last` word. The next line writes the encoded line, and the final line sets `last` to the current input word.

Awk is a fine implementation language for this task: the code is compact and easy to understand. It is also a little slow: about four minutes on a VAX 8550 (a machine with a few MIPS). Throughout the rest of this exercise we’ll ignore execution speed, but first a tempting diversion.

Exercise 4. Implement the `preflen` function in a different language. How does its length and speed compare to the Awk version?

Here is an Awk program to decode the encoded file:

```
{ i = 0 + $1  # int at start of line
  s = substr(s, 1, i)
  print s substr($1, length(" " i))
}
```

A C version isn’t a lot more complicated; you might enjoy writing it.

Table 1 describes the effectiveness of various compression schemes. We see that the prefix length encoding gives a reduction to about 52% of the original size.

Exercise 5. Study the performance of this compression method on your `/usr/dict/words`. Browse the output file, and gather statistics on it.

One of the great things about data compression schemes is that we can ‘‘mix-and-match’’ them. The

alphabet now has 37 characters (26 letters, 10 digits and newline), so we'll have to go from a 5-bit to a 6-bit code. Table 1 shows that if we use the pipeline `preflen | bitcost 6` then we achieve a compression to about 39%.

Both Sides Now

If prefix length encoding is good, then maybe suffix length encoding is even better. The hard way to solve this problem is to write a special-purpose program that sorts the words using right-to-left order and then computes the common suffix length right-to-left. An easier solution is to use the common UNIX program `rev` which reverses the characters on each line.

Exercise 6. Implement `rev` in your favorite language.

We can compress the dictionary with this pipeline

```
rev <dict.in | sort | preflen >cmprsd
```

and expand it with this pipeline:

```
unpreflen <cmprsd | rev | sort >dict.in
```

Table 1 shows that suffix compression gives a reduction to 53.4%, which is a tad worse than prefix compression. At least our stroll down this blind alley was short.

Exercise 7. This pipeline attempts to do both prefix and suffix compression:

```
preflen | rev | sort |
    preflen | bitcost 8
```

Can you spot the problem with it? Characterize the conditions required by the various pipes in this column.

Variable-Length Codes

In a variable-length code, common symbols are assigned short codes. In English, ‘‘e’’ is the most common letter and ‘‘z’’ is the least common. Morse code therefore represents ‘‘e’’ by the succinct ‘‘dot’’ while ‘‘z’’ is ‘‘dash dash dot dot’’. English words represent a kind of variable-length code: common words tend to be short. In Section 6.2.2 of *Sorting and Searching* (described in Further Reading), Knuth presents the 31 most common English words; ‘‘the’’, ‘‘of’’, ‘‘and’’, ‘‘to’’ and ‘‘a’’ (all tiny words) head the list. Long words aren't as frequent; when was the last time you used ‘‘electroencephalographic’’ in a conversation?

A Huffman code represents common symbols by short bit strings and rare symbols by long bit strings. In the original (uncompressed) dictionary, the most common characters are newline (11.56%) and ‘‘e’’ (10.77%), while the least common are ‘‘q’’ (.18%) and ‘‘j’’ (.17%). Figure 1 shows a tree representation of the Huffman code for the letters in that dictionary. A left branch in the tree represents a 0 and a right branch represents a 1. Thus the letter ‘‘e’’ is represented by the 3-bit string ‘‘000’’ while ‘‘j’’

COMPRESSION PIPELINE	Bits	Kbytes	Bits/ Letter	Bytes/ Word	% of Original	Compression Factor
bitcost 8	5534016	691.8	8.00	9.65	100.00	1.00
bitcost 5	3458760	432.3	5.00	6.03	62.50	1.60
preflen ∣ bitcost 8	2862736	357.8	4.14	4.99	51.73	1.93
preflen ∣ bitcost 6	2147052	268.4	3.10	3.75	38.80	2.58
rev ∣ sort ∣ preflen ∣ bitcost 8	2956424	369.6	4.27	5.16	53.42	1.87
hufcost	2959010	369.9	4.28	5.16	53.47	1.87
preflen ∣ hufcost	1611022	201.4	2.33	2.81	29.11	3.44
preflen ∣ subs ∣ bitcost 8	2424784	303.1	3.51	4.23	43.82	2.28
preflen ∣ subs ∣ hufcost	1499540	187.4	2.17	2.62	27.10	3.69
preflen ∣ subs ∣ rmnl ∣ bitcost 8	1851496	231.4	2.68	3.23	33.46	2.99
preflen ∣ subs ∣ rmnl ∣ hufcost	1257113	157.1	1.82	2.19	22.72	4.40
compress ∣ bitcost 8	2421320	302.7	3.50	4.22	43.75	2.29
preflen ∣ subs ∣ rmnl ∣ compress ∣ bitcost 8	1115064	139.4	1.61	1.95	20.15	4.96

Table 1. Performance of the compression pipelines. This table summarizes the techniques applied to a dictionary of 71661 words in 691752 characters (lower-case letters and newlines). Each method is described by its UNIX pipeline, and the first number in each row counts the bits used by the scheme. All other performance measures are derived from the bit count, and give different ways of thinking about the effectiveness of the compression. The last line, for instance, describes a highly compressed file of 139.4 kilobytes. Each character in the dictionary is represented by 1.61 bits (compared to 8 in the original file), and each word in the dictionary is represented in 1.95 bytes (compared to 9.65 originally). The resulting file is 20.15% the size of the original, for a reduction factor of 4.96. The same experiment on a larger dictionary (Webster's Second, which contains 233614 words and 2477182 characters) gave similar results.

is represented by the 9-bit string ‘‘110101000’’. The tree in Figure 1 uses 4.28 bits per letter to encode the original dictionary. Although we applied Huffman codes to bytes, we could just as easily encode letter pairs, syllables, or words in English text. But due to editorial space constraints, I'll have to use the common magazine compression technique of promising to describe Huffman codes in a column all to themselves.

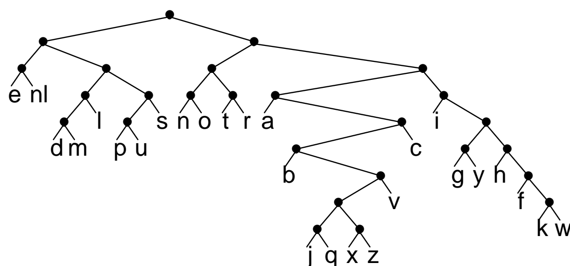

Figure 1. A Huffman code for the dictionary.

I wrote a simple program to measure the cost of Huffman encoding a file; the `hufcost` program is implemented in a few dozen lines of C, Awk and Shell. The Huffman codes themselves required the 2,959,010 bits reported in Table 1. But the `hufcost` program ignores the space required to transmit the Huffman tree itself. When I measured the output of the UNIX `pack` program (originally written by Tom Szymanski at Princeton in 1976; it implements Huffman codes), I found that it used an extra 181 bytes, or one twentieth of a percent more

than my simple program estimated. And Huffman codes combine nicely with prefix-length codes; the pipeline `preflen | hufcost` gives a file about 29% the size of the original.

Common Substrings

When you browse through the prefix-length encoding you notice a lot of words like "5ment" and "6ing". An ancient compression technique represents common suffixes by a succinct code. Modern tools, though, make it easy to search for and to encode common patterns that occur anywhere in a word. I therefore wrote an Awk program to hunt for common substrings in the prefix-length encoded file. Here are the first few lines of its output:

```
7941    ness    2647
6224    ess     3112
5456    nes     2728
4738    ly      4738
4734    ing     2367
4356    ion     2178
4000    ne      4000
3930    in      3930
3921    es      3921
3492    er      3492
    ...
```

The first line states that the substring "ness" occurs 2647 times, and that if we encode it by a single character (for instance, a capital letter), we save $2647 \times (4-1) =$ 7941 characters. That output was made by this Awk program; I'll leave the interpretation as an exercise for Awkophiles.

```
      { n = length($1)
        for (i = 1; i <= n; i++)
            for (j = n+1-i; j > 1; j--)
              cnt[substr($1, i, j)]++
      }
END { for (i in cnt) {
            save = cnt[i] * (length(i)-1)
            if (save > 1000)
                print save "\t" i "\t" cnt[i]
        }
      }
```

The output suggests a number of potentially effective codes, but there is substantial overlap in the list: once we represent "ness" by a single character, there will be far fewer occurrences of "nes" and "ess". I therefore went through the list by hand, and removed substrings with overlap. Here is my final list of 26 substrings:

```
ness ly ing ion er ic te al le
io ous an is at ity ed ia or
ment st ra ou tic abl all ine
```

I then ran the pruned file through a short Awk program to

write the following `subs` program:

```
{   gsub(/ness/, "A")
    gsub(/ly/,   "B")
    gsub(/ing/,  "C")
     ...
    print
}
```

Table 1 shows the effectiveness of this encoding in several different pipelines.

Exercise 8. Write the program that writes `subs`. Some readers will prefer to translate into `sed` code rather than Awk. Experiment with numbers of substrings other than 26.

Out, Damned New Line!

After we use the prefix encoding, each line in the file consists of numbers followed by letters followed by a newline. The newline character is redundant, so we can remove it with the `rmnl` program, which is implemented as

```
tr -d '\012'
```

The most intuitive way for me to measure this change is to see that it drops the length of the output of `preflen | subs` from 4.23 to 3.23 characters per word.

Exercise 9. Write the program to re-insert newlines.

Lempel-Ziv Compression

The UNIX `compress` command implements the compression method first described by Abraham Lempel and Jacob Ziv. While Huffman codes represent fixed-length strings with variable-length codes, Lempel-Ziv methods represent variable-length strings with fixed-length codes. As an example of another powerful data compression technique, I am not going to describe Lempel-Ziv coding, but merely provide a pointer to Bob Lucky's description under Further Reading. Table 1 shows the effectiveness of `compress` in several different pipelines: by itself, it gives a factor of 2.29, but combined with the pipeline `preflen | subs | rmnl`, it squeezes the dictionary by a factor of almost 5.

Exercise 10. Find out what compression programs are available on your system, and measure their effectiveness on some of your files.

The Bottom Line

Our little experiment in data compression is not atypical of the field. Several methods gave us a compression factor of two. When we mixed-and-matched those, we got up to three. With a whole lot of work and a little engineering common sense, we ended up with a factor near five. But we've just barely scratched the surface of this problem: the sidebar on page PPP describes two other

programs for the task, and here are a few exercises you can try yourself.

Exercise 11. Try squeezing a dictionary on your system; measure the effect of various methods. For our experiment we ignored upper-case letters; how would you deal with them? What about languages other than English?

Principles

Data Compression. All of the data compression techniques we used are applicable to many problems beyond dictionaries: differential coding, Huffman codes, abbreviating common patterns, removing redundant characters, and Lempel-Ziv coding. We built some special-purpose programs for this job, but whenever possible we used general UNIX tools, such as `rev`, `pack` and `compress`.

Prototyping on UNIX. Bob Martin of Bellcore observes that ''You're always going to build a prototype — the only question is whether you're going to deliver it to the customer.'' In this exercise, our prototypes only measured the effectiveness of compression schemes; we didn't always implement the encoding and decoding programs. Whenever possible we used existing UNIX tools, and we connected them in pipelines.

The Structure of Computing. You'll probably never have to compress a dictionary, but the lessons of this exercise apply to many programming tasks. Good programmers solve problems the way we compressed the dictionary: they hunt for patterns.

Further Reading

Bob Lucky's *Silicon Dreams: Information, Man and Machine* was published by St. Martin's Press (NY, NY) in 1989. It is a delightful introduction to many topics in information theory and computing, including the generation, transmission and processing of text, speech and pictures. Chapter 3 provides an excellent introduction to Huffman codes and the Lempel-Ziv algorithm. And as an extra bonus, each chapter ends with one of Lucky's thought-provoking *Reflections* columns that originally appeared in *IEEE Spectrum*.

Any discussion of programming must acknowledge Don Knuth's monumental *Art of Computer Programming*, published by Addison-Wesley. The seven volumes (three of which have appeared so far) directly discuss much of computer science, and sneak in most of the rest. Huffman codes, for instance, are described in Section 2.3.4.5 of *Volume 1: Fundamental Algorithms* (1968) as an application of tree path lengths. In Section 6.1 of *Volume 3: Sorting and Searching* (1973), Knuth studies the fascinating problem of representing a file of all prime numbers less than one million. He uses a number of techniques for this problem, including differential coding (storing the differences between successive primes) and

variable-length codes.

Exercise 12. Sketch how you might compress a file of the primes less than one billion.

In Section 6.3 of volume 3, Knuth considers a variant of our problem: a compressed dictionary that allows rapid lookup. He sketches several fascinating approaches to prefix and suffix analysis, and an elegant interpretive language for representing dictionaries.

Every time I return to Knuth's books, I leave with fresh new insights into computing.

Solutions to Selected Exercises

3. The first argument to this Awk implementation of `bitcost` is the number of bits per character; an optional second argument may specify a file name:

```
BEGIN { bits = ARGV[1]
        ARGV[1] = ""
        if (ARGC == 2) ARGC--
      }
      { c += length($0) + 1 }
END   { print c * bits }
```

The messy `BEGIN` action reads the number of bits per character and ensures that later files (or the standard input) will be processed properly. The second pattern counts the characters in the file, and the `END` pattern prints the number of characters times the number of bits per character.

4. This C program implements the prefix-length compression algorithm `preflen`:

```
#include <stdio.h>
main()
{   char word[100], last[100];
    int i;
    strcpy(last, "");
    while (scanf("%s", word) != EOF) {
        for (i=0; word[i]==last[i]; i++)
            ;
        printf("%d%s\n", i, &word[i]);
        strcpy(last, word);
    }
}
```

This C program takes just 14 seconds on the dictionary, for a speedup factor of 17 over the Awk version. (This is an unusually bad ratio because the Awk program creates and destroys a string for every call to `substr`.)

5. This table gives the histogram of the prefix lengths in the prefix-length encoded file:

0	26	11	1753
1	318	12	873
2	2622	13	491
3	9493	14	226
4	14011	15	106
5	12315	16	38
6	9601	17	14
7	7248	18	4
8	5505	19	5
9	4096	20	0
10	2914	21	2

The 26 0-length prefixes are for the letters of the alphabet, while the two 21-length prefixes were for adding the “-ic” and “-y” suffixes to “electroencephalograph”.

6. Here is a C implementation (less declarations) of the `rev` program:

```
while (scanf("%s", thisword) != EOF) {
     for (i=strlen(thisword)-1; i>=0; i--)
          putchar(thisword[i]);
     putchar('\n');
}
```

8. This Awk program wrote `subs`:

```
BEGIN  { x = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
         printf "{"
       }
       { print "\tgsub(/" $2 "\/,\t\"" \
                substr(x, NR, 1) "\")"
       }
NR==26 { print "\tprint\n}"; exit }
```

11. If we allow capital letters in the dictionary, then our 5-bit code expands to 6 bits for a 20% increase. Huffman codes exploit the rarity of capital letters (4045 capitals in 72275 words on this dictionary) for an increase of about one percent. If we instead insert “!” before each capital letter, that is about half a percent increase. In their program described in the sidebar on page PPP, Morris and Thompson exploited the fact that all capital letters are at the beginning of words to encode case by two different newline characters.

12. Knuth observes that we can encode the primes less than N in N bits by sending a bit for each integer telling whether it is prime; he then squeezes a factor of two from this method by sending only the bits for odd integers. We can get down to about $N/3$ bits by ignoring the integers congruent to 0, 2, 3 and 4 mod 6. If we consider the integers mod $2\times3\times5$, then we get down to about $8/30$ of the original N bits. But before we pursue this method too far, we should consider an insight I first learned from Ed McCreight of Adobe Systems Incorporated: the most succinct representation of a set of prime numbers is a program to compute them.

Sidebar — Some Other Big Squeezes

The problem of squeezing a dictionary has been around for a long time. In 1974, Robert Morris and Ken Thompson wrote an internal Bell Labs memo on ‘‘Webster’s Second on the Head of a Pin’’. They first compressed common prefixes, and then encoded several common suffixes. Finally, they used a variable-length code based on four-bit hexadecimal digits. Their final compression ratio was 4.52.

After he read a draft of this column, Peter Weinberger of Bell Labs was able to squeeze the dictionary used in our exercise down to 1.46 bits per byte. He writes: ‘‘For each of the 71661 words, find its length, and the length of the prefix it has in common with the previous word (but never use more than 9 common characters). Form file A of 71661 bytes by multiplying, for each line, the first of the two numbers by 10, and adding the second number. This fits in a byte, and `compress` drops this file to 31571 bytes. (There’s a cheat. There is exactly one word longer than 23 bytes, which has length 45. We’ll represent its length by 24, at the cost of a slightly longer program.)

‘‘File B consists of the dictionary with the newlines and common prefixes (of size no more than 9) removed. It is 208094 bytes, and `compress` shrinks it to 94531. The total is 94531+31571=126102 bytes, or about 1.459 bits/character, for a compression factor of 5.49.’’

An Extra Bonus for Readers of the Draft!

This table will not be included in the final version, but is here for your amusement and edification.

COMPRESSION PIPELINE	Bits	Kbytes	Bits/ Letter	Bytes/ Word	% of Original	Compression Factor
cat	bitcost 8	19817456	2477.2	8.00	10.60	100.00	1.00
cat	bitcost 5	12385910	1548.2	5.00	6.63	62.50	1.60
preflen	bitcost 8	10019552	1252.4	4.04	5.36	50.56	1.98
preflen	bitcost 6	7514664	939.3	3.03	4.02	37.92	2.64
rev	sort	preflen	bitcost 8	9850832	1231.4	3.98	5.27	49.71	2.01
cat	hufcost	10610861	1326.4	4.28	5.68	53.54	1.87
preflen	hufcost	5700113	712.5	2.30	3.05	28.76	3.48
preflen	subs	bitcost 8	8565304	1070.7	3.46	4.58	43.22	2.31
preflen	subs	hufcost	5381767	672.7	2.17	2.88	27.16	3.68
preflen	subs	rmnl	bitcost 8	6696392	837.0	2.70	3.58	33.79	2.96
preflen	subs	rmnl	hufcost	4570220	571.3	1.84	2.45	23.06	4.34
compress	bitcost 8	8141368	1017.7	3.29	4.36	41.08	2.43
preflen	subs	rmnl	compress	bitcost 8	4087624	511.0	1.65	2.19	20.63	4.85

Table 2. Performance on Webster’s Second. This table uses the same format as Table 1. It shows the performance on a dictionary of 233614 words and 2477182 characters.

Jon Bentley is a Member of Technical Staff in the Computing Science Research Center at AT&T Bell Laboratories.