literate programming

Moderated by
Christopher J. Van Wyk

PRINTING COMMON WORDS

Moderator's Introduction to Column 1

Two Programming Pearls columns in 1986 presented literate programs by Donald Knuth. The May 1986 column contained a WEB program to generate a sorted list of M distinct numbers randomly chosen from 1 through N. Programming Pearls of April 1987 contained readers' comments on Knuth's literate program, as well as an unimplemented solution to the problem by David Gries. The June 1986 Programming Pearls presented Knuth's WEB program to print the k most common words in a file, in decreasing order by frequency. That column also elicited comments from many readers.

Several readers wondered how well a small example can serve to illustrate the features of a system for literate programming. The issue arose in two very different ways: One reader who thought WEB worked acceptably in the small doubted that it would serve a large project well; another suggested that the WEB solution's apparent clumsiness in the small could be attributed to its being designed for larger, more complex programs.

A couple of readers very much appreciated Doug McIlroy's addition of a figure to the documentation. They also thought that trie-sorting in Sections 37–39 needed more comments than "The restructuring operations are slightly subtle here."

One sharp-eyed reader noticed that, although Section 38 uses "the fact that `count[0] = 0`," `count[0]` is never initialized. Although some Pascal compilers detect references to uninitialized variables, apparently the compiler that Knuth used at SAIL does not.

In this inaugural column, we present another solution to the problem of printing the k most common words in a file. The author of this solution is David Hanson, a professor of computer science at Princeton University and an editor of the journal *Software—Practice & Experience*. Hanson's program illustrates nicely the use of abstract data types on the way to the solution of a problem. It discusses the role of profiles of execution in the design of good programs, a topic that is also discussed in this month's Programming Pearls. Hanson's solution shows how one can design a system for literate programming that involves much less machinery than WEB. Finally, note that Hanson solved a slightly different problem than Knuth; although that makes little difference to our discussion of literate programs, it highlights the importance of careful problem specification in the design of large systems.

Printing Common Words

1. Introduction. In describing Don Knuth's WEB system in one of his "Programming Pearls" [*Communications of the ACM 29*, 5 (May 1986), 364–369], Jon Bentley "assigned" the following programming problem: "Given a text file and an integer k, you are to print the k most common words in the file (and the number of their occurrences) in decreasing frequency."

It is unclear from this problem statement what to do with "ties," that is, does k refer to words or word frequencies? For example, in the problem statement, "the" occurs three times, "k," "in," "and," and "file" each occurs twice, and the rest of the words each occurs once. If the program is invoked with the statement as input and $k = 2$, which word should be output as the second most common word? A rephrasing of the problem removes the ambiguity: "Given a text file and an integer k, you are to print the words (and their frequencies of occurrence) whose frequencies of occurrence are among the k largest in order of decreasing frequency."

Using this problem statement, the output of the program with the original problem statement as input and with $k = 2$ is

```
3 the
2 file
2 and
2 in
2 k
```

This work was supported in part by the National Science Foundation under Grant MCS-8302398.

Bentley posed this problem to present a "real" example of WEB usage. For more information about WEB, see Knuth's "Literate Programming," *The Computer Journal 27*, 2 (May 1984), 97–111. Knuth's solution appears in *Communications of the ACM 29*, 6 (June 1986), 471–483, along with a review by Doug McIlroy.

The solution given here is written in the C programming language and presented using the `loom` system to generate the printed program and its explanation. `loom` is a preprocessor whose input is a text file with embedded references to fragments of the program. `loom` retrieves these fragments, optionally pushes them through arbitrary filters, and integrates the result into the output.

`loom`'s output is usually input to a document formatter, such as `troff` or TEX. `loom` was originally written by Janet Incerpi and Robert Sedgewick and used in preparation of Sedgewick's book *Algorithms* (Addison-Wesley, Reading, Mass., 1983). Starting from their program, I rewrote `loom` for use in writing a book and several papers.

`loom` is not as ambitious or as comprehensive as WEB. It does, however, have the virtue of independence from both formatting and programming languages. It does not, for example, provide the comprehensive indexing, cross-referencing, or pretty printing facilities of WEB. With help from its associated filters, `loom` does provide indexing of the identifiers used in the program fragments, although the index is omitted here for brevity. And since it is not necessary to present the whole program, irrelevant details can be omitted permitting the documentation to concentrate on the important aspects of the programs. I have formatted this program description in a style similar to WEB for comparison purposes, but the formatting of `loom`'s output is not constrained to any one style. Using `loom` also has an effect similar to WEB: Developing and writing about programs concurrently affect both activities dramatically.

2. Definitions. The problem statement does not give a precise definition of a "word" or of the details of program invocation. *Words* are given by the set $\{w \mid w = aa^*$ and $|w| \leq 100\}$ where $a \in \{\mathtt{a} \cdots \mathtt{z}, \mathtt{A} \cdots \mathtt{Z}\}$; that is, a word is a sequence of one or more upper- or lowercase letters, up to a maximum of 100 letters. Only the first 100 characters are considered for words longer than 100 characters.

The program, called `common`, is invoked with a single optional argument that gives the value of k and reads its input from the standard input file. If the argument is omitted, the value of the environment variable `PAGESIZE` is used; the default is 22.

3. The main program. As suggested in *Software Tools* by Kernighan and Plauger (Addison-Wesley, Reading, Mass., 1976), the structure of the program can often be derived from the structure of the input data. The input to `common` is a sequence of zero or more words, which suggests the following structure for the main program:

```
/* initialize k */
/* initialize word table */
while (getword(buf, MAXWORD) != EOF)
   addword(buf);
printwords(k);
```

where `buf` is a character array of `MAXWORD` characters, and `getword` places the next word in the input in `buf` and returns its length, or `EOF` at the end of file. `MAXWORD` is defined to be 101 to allow room for a terminating null character. `addword` adds the word in `buf` to the table of words, and `printwords(k)` prints the words with the k largest frequencies.

Getting program arguments and environment variables, such as k and `PAGESIZE`, is a common feature of many UNIX® programs and the code is idiomatic. Examples can be found in *The UNIX Programming Environment* by B. W. Kernighan and R. Pike (Prentice-Hall, Englewood Cliffs, N.J., 1984).

4. Reading words. `getword` reads the next word from the input. This is accomplished by discarding characters up to the next occurrence of a letter, then gathering up the letters into the argument buffer:

```
int getword(buf, size)
char *buf;
int size;
{
   char *p;
   int c;

   p = buf;
   while ((c = getchar()) != EOF)
      if (isletter(c)) {
         do {
            if (size > 1) {
               *p++ = c;
               size--;
               }
            c = getchar();
            } while (isletter(c));
         *p = '\0';
         return p - buf;
         }
   return EOF;
}
```

`size` is compared with 1 to ensure that there is room for the terminating null character. `isletter` is a macro that tests for upper- or lowercase letters:

```
#define isletter(c) (c >= 'a' && c <= 'z' || \
     c >= 'A' && c <= 'Z')
```

5. Storing the words. The words must be stored in a table along with the number of times they occur in the input. This table must handle two kinds of access: While the input is being read, the table is "indexed" with a word in order to increment its frequency count. After the input has been read, the entries with the k largest fre-

UNIX is a registered trademark of AT&T Bell Laboratories.

quency counts must be located and printed in decreasing order of those counts.

These two kinds of access are disjoint; that is, initially, all accesses to the table are of the first kind, followed by only accesses of the second kind. Consequently, the table representation can be designed to facilitate the first kind of access, and then *changed* to facilitate the second.

A hash table is appropriate for indexing the table with words. Since the size of the input is unknown, a hash table in which collisions are resolved by chaining is used. Space for both the word and the table entry can be allocated dynamically. The hash table itself, `hashtable`, is an array of pointers to `word` structures:

```
#define HASHSIZE 07777  /* hash table size */
struct word {
   char *word;          /* the word */
   int count;           /* frequency count */
   struct word *next;   /* link to next entry */
} *hashtable[HASHSIZE+1];
```

The bounds of `hashtable` are 0 to $2^n - 1$, where n is 12 here. Using a power of two facilitates rapid computation of the index into `hashtable` given a hash number: If `h` is a hash number, the index is `h&HASHSIZE`. `hashtable` is initialized in `main` to `NULL` pointers.

6. `addword(buf)` adds the null-terminated string in `buf` to `hashtable`, if necessary, and increments its `count` field. To compute the index into `hashtable`, the contents of `buf` must be "hashed" to yield a hash number `h`, from which the index is computed as described above. A simple yet effective hash function is to sum the codes of the characters in `buf`. This function also yields the length of the word, which is needed to add new words to the table. Putting this all together produces `addword`:

```
addword(buf)
char *buf;
{
   unsigned int h;
   int len;
   char *s, *alloc();
   struct word *wp;

   h = 0;      /* compute hash number of buf[1..] */
   s = buf;
   for (len = 0; *s; len++)
      h += *s++;
   for (wp = hashtable[h&HASHSIZE]; wp;
      wp = wp->next)
         if (strcmp(wp->word, buf) == 0)
            break;
   if (wp == NULL) {        /* a new word */
      wp = (struct word *) alloc(1, sizeof *wp);
      wp->word = alloc(len + 1, sizeof(char));
      strcpy(wp->word, buf);
      wp->count = 0;
      wp->next = hashtable[h&HASHSIZE];
      hashtable[h&HASHSIZE] = wp;
      total++;
      }
   wp->count++;
}
```

`addword` also increments a global integer, `total`, which counts the number of distinct words in the table. This number is required in the second phase of the program. `strcmp` is a C library function that returns 0 if its two arguments point to identical strings, and `strcpy` is a C library function that copies the characters in its second argument into its first.

`alloc(n, size)` allocates space for `n` contiguous objects of `size` bytes each by calling `calloc`, a C library function that does the actual allocation and clears the allocated space. `alloc`'s primary purpose is to catch allocation failures. Many C programmers erroneously assume that `calloc` cannot fail. On machines like the VAX, allocation rarely fails, but on smaller machines, failure is common.

7. Printing the words. As suggested in the outline for `main`, given above, `printwords(k)` prints the desired output. To print the k most common words as specified, `printwords` must sort the contents of the table in decreasing order of the `count` values, and print the first k entries. Since the frequencies range between 1 and N, where N is the number of words, sorting them can be accomplished in time proportional to N (assuming everything fits into memory) by allocating an array of pointers to `words` that is indexed by the frequency of occurrence. Each element in the array points to the list of `words` with the same `count` values; that is, `list[i]` points to the list of `words` with `count` fields equal to `i`.

```
printwords(k)
int k;
{
   int i, max;
   struct word *wp, **list, *q;

   list = (struct word **) alloc(total, sizeof wp);
   max = 0;
   for (i = 0; i <= HASHSIZE; i++)
      for (wp = hashtable[i]; wp; wp = q) {
         q = wp->next;
         wp->next = list[wp->count];
         list[wp->count] = wp;
         if (wp->count > max)
            max = wp->count;
         }
   for (i = max; i >= 0 && k > 0; i--)
      if ((wp = list[i]) && k-- > 0)
         for ( ; wp; wp = wp->next)
            printf("%d %s\n", wp->count, wp->word);
}
```

`max` keeps track of the largest frequency count, which is usually much less than N, and provides a starting point for the reverse scan of `list`.

8. Performance. Bentley did not give specific performance criteria for `common`, but he did say that "a user should be able to find the 100 most frequent words in a twenty-page technical paper without undue emotional trauma." To test `common` I concatenated seven of the documents from volume 2 of the *UNIX Programmer's Manual* from the Berkeley 4.2 UNIX system to form a

test file with 11,786 lines, 47,878 words (by common's definition of "word"), 4,149 of which are unique, and 275,516 characters. (The documents were the descriptions of awk, efl, the UNIX implementation, the UNIX i/o system, lex, sccs, and sed.)

common with $k = 0$ and this test file as input took 4.6 s on a VAX 8600 running Berkeley 4.3 UNIX. By way of comparison, consider the following program, called charcount:

```
main()
{
   int c, n = 0;

   while ((c = getchar()) != EOF)
      n++;
   printf("%d\n", n);
}
```

charcount is about the minimum "interesting" program in this class of programs, and its execution time gives a measure of the cost of simply reading the input. With the test file as input, charcount ran in 0.9 s. The ratio of the speed of common to charcount, which is independent of machine dependencies such as CPU speed and I/O costs, is 5.11. Thus, using the implementation of common described above, finding the k most common words costs approximately five times as much as just counting the characters.

9. Improvements. To investigate the propects for improving the execution speed of common, I profiled its execution with gprof [S. L. Graham, P. B. Kessler, and M. K. McKusick, "An Execution Profiler for Modular Programs," *Software—Practice & Experience 13*, 8 (Aug. 1983), 671–685]. gprof takes profiling data produced by executing the program and generates a report detailing the cost of each function and its dynamic descendants.

These measurements revealed that addword and its descendants accounted for 62 percent of the execution time. For example, strcmp was called 144,219 times and accounted for 21 percent of the *total* execution time. strcmp was the most frequently called function. getword accounted for 32 percent of the execution time, and the other functions accounted for the remaining 6 percent.

10. The cost of strcmp can be reduced two ways: doing fewer comparisons and putting the code in-line. To do the string comparison in-line, the if statement in addword in which strcmp is called is replaced by

```
for (s1 = buf, s2 = wp->word; *s1 == *s2; s2++)
   if (*s1++ == '\0') {
      wp->count++;
      return;
      }
```

and the remainder of addword is revised accordingly. This change reduced the running time by 10.8 percent to 4.56 charcounts (4.1 s).

The number of string comparisons can be reduced by storing additional information with each word that is checked before the string comparison is undertaken. For example, the hash number for each word can be stored in a hash field, and only those words for which wp->hash is equal to h are actually compared to buf. I tried this improvement, and it *increased* the running time to 5 charcounts. I also tried storing and comparing the lengths instead of the hash numbers, and the result was the same.

11. The test input has 4,149 different words, which is slightly larger than the size of hashtable (4,096). With a hash table size of 512 and the improvements described above, the running time increased to 5.56 charcounts (5 s). gprof showed that 66 percent of the time was spent in addword, 29 percent in getword, and 5 percent elsewhere.

The time spent searching the hash chains would be reduced if the most common words were near the front of the chains. This effect can be accomplished by using the "move-to-front" heuristic: Each time a word is found, it is moved to the front of its hash chain. This heuristic can be incorporated into addword by adding a pointer that "follows" wp down the chain:

```
addword(buf)
char *buf;
{
   unsigned int h;
   int len;
   char *s, *s1, *s2, *alloc();
   struct word *wp, **q, **t;

   h = 0;      /* compute hash number of buf[1..] */
   s = buf;
   for (len = 0; *s; len++)
      h += *s++;
   t = q = &hashtable[h%HASHSIZE];
   for (wp = *q; wp; q = &wp->next, wp = wp->next)
      for (s1 = buf, s2 = wp->word; *s1 == *s2; s2++)
         if (*s1++ == '\0') {
            wp->count++;
            if (wp != *t) {
               *q = wp->next;
               wp->next = *t;
               *t = wp;
               }
            return;
            }
   wp = (struct word *) alloc(1, sizeof *wp);
   wp->word = alloc(len + 1, sizeof(char));
   strcpy(wp->word, buf);
   wp->count = 1;
   *q = wp;
   total++;
}
```

This change reduced the running time with a hash table size of 512 to 4.56 charcounts (4.1 s)—equal to that of the time for a hash table size of 4,096 without the heuristic. Using a hash table size of 4,096 *and* the move-to-front heuristic, the running time was 4.5 charcounts (4 s). This last measurement verifies that the heuristic does not impair performance when the size of the input is less than the hash table size, which is not obvious from

the code. For other applications of the move-to-front heuristic, see J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A Locally Adaptive Data Compression Scheme," *Communications of the ACM 29*, 4 (Apr. 1986), 320–330, and the references therein.

12. Identifying the common words in a 47,878-word file in 4.5 `charcounts` seemed fast enough to avoid "undue trauma." Nevertheless, I wondered if the standard UNIX macros for testing character classes were significantly faster than the `isletter` macro above. The standard macros use table lookup and bit testing, which could be faster than the explicit comparisons used in `isletter`.

This change reduced the running time by 8 percent to 4.1 `charcounts` (3.7 s). I made the change by including the standard header file and by defining `isletter` to be `isalpha`. On UNIX systems where `EOF` is not a valid argument to the `isalpha`, `isletter` should be defined as `(c != EOF && isalpha(c))`. Both definitions gave the same timings.

`gprof` indicated that in this *final* version of `common`, `addword` and its descendants took 54 percent of the time, `getword` took 39 percent, and everything else took the remaining 7 percent. On behalf of `addword`, `alloc` and its descendants accounted for 11 percent of the time, so allocation accounts for about 20 percent of the cost of `addword`. By preallocating some space at compile time, this cost might be reduced by half, but this change would yield only a 5 percent speedup, so it was not attempted.

The changes made to improve `common`'s performance were made as *additions* to the program, and conditional compilation is used to select the "fast" version. Thus both the program and this document describe not only the initial program, but also trace its evolution.

13. Development notes. Writing `common` and this documentation, which was done concurrently, took about 9.5 hours. The initial 5 hours included a false start: The first version sorted the words by making `list` (in function `printwords`) an array of pointers to `words` and calling the C library function `qsort` to sort them. This version ran in 11.22 `charcounts` (10.1 s), and I spent another 2.5 hours making measurements and improvements. Ultimately, I reduced the running time to 4.5 `charcounts` by replacing the general `qsort` (which calls a function for every comparison and took over 50 percent of the time) with one written specifically for sorting an array of pointers to `words`, and by applying the improvements described above.

Chris Fraser and I observed that the frequency counts were in the range 1 to N, and he suggested the rather obvious linear-time radix sort (with radix $N + 1$) described above. Indeed, final measurements show that `printwords` takes only 1 percent of the time. I spent the other 4.5 hours revising the program and this explanation and rerunning the performance measurements.

14. Typical `loom` usage involves the document file and the program files (e.g., `common.lo` and `common.c`). The document file contains references to fragments in the program files. `loom` combines these into a TEX input file (e.g., `common.tex`), which is typeset by TEX.

For small programs, such as `common`, the document and program files can be combined into a single file; for `common` both are combined into `common.c`. C conditional compilation facilities are used to remove the document part when `common.c` is compiled, and `loom` processes `common.c` to form `common.tex`, obtaining the code fragments from `common.c`. Thus a single file contains both the program and its explanation, making `loom`'s usage similar to `WEB`'s.

David R. Hanson
Department of Computer Science
Princeton University
Princeton, NJ 08544

Review of Hanson's Solution

[*John Gilbert is an associate professor of computer science at Cornell University. His insightful comments on programs have impressed me for many years; this review shows that he has not lost his touch.*]

What is a literate program, anyway? A program, like an essay, can be written for many purposes: to teach, to learn, to experiment with new forms, to solve an immediate problem, to sell refrigerators, or to sell workstations. The solutions to Jon Bentley's problem by Knuth, McIlroy, and Hanson are all "literate programs," but they differ in style to a startling degree.

Architecture may be a better metaphor than writing for an endeavor that closely mixes art, science, craft, and engineering. "Put up a house on a creek near a waterfall," we say, and look at what each artisan does: The artist, Frank Lloyd Wright (or Don Knuth), designs Fallingwater, a building beautiful within its setting, comfortable as a dwelling, audacious in technique, and masterful in execution.

Doug McIlroy, consummate engineer, disdains to practice architecture at all on such a pedestrian task; he hauls in the pieces of a prefabricated house and has the roof up that afternoon. (After all, his firm makes the best prefabs in the business.) David Hanson puts together a roomy and craftsmanlike cabin, probably closest of the three to the spirit of the original commission.

The programs differ not only in their proportions of art, engineering, and craftsmanship, but in their very intent. Of course the authors wrote them to solve a set problem, but each seems to have had a different purpose in mind. Knuth's is art and exposition. McIlroy's solves a one-shot problem. Hanson's could be a piece of

real-world software, perhaps inside some unusual word processing system.

Hanson first chooses his floor plan and building materials—that is, an algorithm and data structures. Craftsmanlike, he balances efficiency, clarity, and the natural use of available materials. Some analysis (which he could have made even more explicit than he did for the benefit of the apprentice carpenter) points the way: Say the input is n words of which d are different, and the output is to be k words. We expect k to be much smaller than d, and d to be much smaller than n. The necessary abstract data type is a multiset of words, supporting the operations "Insert" and "List" (in order of decreasing frequency), with a sequence of Inserts preceding the single List. Space is at a premium in large text processing applications, so the data structure should have size $O(d)$ rather than $O(n)$. Then Insert should be as fast as possible, since Insert alone needs to happen n times. List need only deal with d items.

Given these constraints, hashing for Insert and sorting for List are appropriate choices of well-tried materials. Hanson's linear-time bucket sort for the frequencies in List is an elegant final touch. Notice the different trade-offs in the three programs: Knuth expends a good deal of effort to introduce a novel data structure for efficiency; McIlroy uses the fastest and most easily built Insert of all, namely, appending to a sequence, but the cost is a data structure of size $O(n)$.

After these basic choices are made, the house is well laid out, and the rooms are planned harmoniously—the program is cleanly divided into manageable bites of code and documentation. The word processor's maintainer will be glad the architect has found a "balance between formal and informal exposition so that a reader can grasp what is happening without being overwhelmed by detail" [1]. Here the `loom` system seems to provide considerable support, though Hanson has too little space to show us as much about it as we might wish.

Inside the individual rooms, the architect is surprisingly careless in his choice of floors and moldings. Many variable names are misleading or uninformative: The parameter "`size`" to `getword` is used locally to measure space left in a buffer, "`total`" violates the rule that a global variable should have the most descriptive name possible (it's the *total* number of different words encountered), and the names "`p`" and "`c`" in `getword`, "`h`" and "`s`" and "`wp`" in `addword`, and "`i`" and "`q`" in `printwords` are content free and sometimes undocumented. The text contains no mention of either the fact or the reason that `getword` returns the length of the word as its value; I suspect this window frame was borrowed from another house without being planed to fit properly. (It also requires that `EOF` not happen to be a positive integer smaller than `size`.) The eight English words of comments within the C code should either have been expanded into a reasonable commenting style or left out on the grounds that the textual documentation makes them unnecessary.

To Hanson's credit, these stylistic infelicities are all confined to pieces of code small enough to be puzzled out without too much trouble, but they would keep me from using the program as an example of style in a beginning computer science course. The move-to-front version of `addword` in Section 11 goes farthest over the line between the concise and the cryptic.

A program to solve the common-words problem must read its input from someplace and convert the resulting string of characters into a sequence of words. This is the plumbing of a text processing program—usually ugly, never interesting to discuss, and always the first thing that clogs up. (Unclogging the plumbing took half the effort I spent on my solution to this problem, though that's partly because I was learning Common Lisp at the time.) McIlroy's plumbing is my favorite, since the natural idiom for transforming a stream of data from one type to another is a coroutine rather than a subroutine, and UNIX filters and pipes do that perfectly. (The ugly part is the quote marks on two adjacent lines that mean a newline character.) Hanson's plumbing is not beautiful, but he does a good job of isolating it from the rest of the program. The owner can ignore the plumbing until it needs work, and then fix it without tearing down the walls.

Literacy in programming means different things in different circumstances. It's not a matter of artistry or efficiency alone; it's more a question of suitability in context. Knuth's expository gem will teach future readers about programming style and data structures, whether they use the code or not. McIlroy's six liner is not itself an enduring piece of work, but it is a clear example of how to use an enduring set of tools. Hanson's real-world code, then, must be evaluated according to whether it is robust, flexible, and easy to maintain. Despite roughness in low-level style, the program meets these goals well. Hanson demonstrates that "literate programming" is a viable approach to creating works of craft as well as works of art.

REFERENCE

1. Knuth, D.E. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

John Gilbert
Computer Science Department
Upson Hall
Cornell University
Ithaca, NY 14853

For Correspondence: Christopher J. Van Wyk, AT&T Bell Laboratories, Room 2C-457, 600 Mountain Avenue, Murray Hill, NJ 07974.