Appeared in Communications of the ACM 30, 7 (July 1987), 594-599.

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 occur twice, and the rest of the words each occur 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
```

Bentley posed this problem to present a "real" example of `WEB` usage. For more information about `WEB`, see D. E. Knuth, 'Literate Programming', *The Computer Journal 27*, 2 (May 1985), 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 papers.

`loom` is not as ambitious or as comprehensive as `WEB`. It does, however, have the virtue of independence from

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

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 affects both activities dramatically.

2. Definitions. The problem statement does not give a precise definition of a "word" nor of the details of program invocation. *Words* are given by the set $\{w \mid w = aa^* \text{ and } |w| \leq 100\}$ where $a \in \{\texttt{a} \cdots \texttt{z}, \texttt{A} \cdots \texttt{Z}\}$; i.e., a word is a sequence of one or more upper- or lower-case 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`, are common features 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, 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 frequency 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, 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 2 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++;
  wp = hashtable[h&HASHSIZE];
  for ( ; 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 of 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 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 `word`s that is indexed by the frequency of occurrence. Each element in the array points to the list of `word`s with the same `count` values, that is, `list[i]` points to the list of `word`s 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 seconds 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 seconds. 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 descendents.

These measurements revealed that `addword` and its descendents 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 changed reduced the running time by 10.8 percent to 4.56 `charcount`s (4.1 secs.).

The number of string comparisons can be reducing 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 `charcount`s. 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` (4096). With a hash table size of 512, and the improvements described above, the running time increased to 5.56 `charcounts` (5 secs.). `gprof` showed that 66 percent of 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 secs.)—equal to that of the time for a hash table size of 4096 without the heuristic. Using a hash table size of 4096 *and* the move-to-front heuristic, the running time was 4.5 `charcounts` (4 secs.). This last measurement verifies that the heuristic doesn't impair performance when the size of the input is less than the hash table size, which isn't 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), 320–330, Apr. 1986, and the references therein.

12. Identifying the common words in a 47,878-word file in 4.5 `charcount`s 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 `charcount`s (3.7 secs.). 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 descendents took 54 percent of the time, `getword` took 39 pecent, and everything else took the remaining 7 percent. On behalf of `addword`, `alloc` and its descendents accounted for 11 percent of the time, so allocation accounts for about 20 percent of the cost of `addword`. By pre-allocating some space at compile-time, this cost might be reduced by half, but this change would yield only a 5 percent speed up, so it wasn't 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 `printwords`) an array of pointers to `word`s and calling the C library function `qsort` to sort them. This version ran in 11.22 `charcount`s (10.1 secs.) and I spent another 2.5 hours making measurements and improvements. Ultimately, I reduced the running time to 4.5 `charcount`s 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 `word`s, 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 a radix of $N + 1$) 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 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

ERRATA: "Printing Common Words", *Communications of the ACM 30*, 7 (July 1987), 594–599. (Also appears as *Printing Common Words*, Tech. Report 86-18, Dept. of Computer Science, The Univ. of Arizona, Tucson, May 1986.)

Several readers found an error in the common words program presented in the "Literate Programming" column and others have suggested improvements.

The error, pointed out by Michael Shook and others, is in allocating `list` in `printwords`: it's potentially too small. The original intention was to allocate N entries in `list`, where N is the number of words in the input. However, only `total` entries are allocated, and `total` is the number of *unique* words in the input. If `total` is less than the maximum word frequency, `list` is indexed erroneously. This situation probably occurs infrequently since `total` is usually much larger than the maximum frequency for most "normal" inputs. The input `hello hello` demonstrates the problem and causes `common` to fail.

The error can be fixed by making `total` count the number of words in the input, which can be done by moving the statement `total++` from `addword` into the loop that calls `addword` in `main`. A better solution, however, is to eliminate `total` and make `list` just large enough to accommodate the largest frequency of occurrence, which can be done in `printwords` by making a pass over the hash table to compute the largest frequency. This version of `printwords` is

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

  max = 0;
  for (i = 0; i <= HASHSIZE; i++)
    for (wp = hashtable[i]; wp; wp = wp->next)
      if (wp->count > max)
        max = wp->count;
  list = (struct word **) alloc(max + 1, sizeof wp);
  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;
      }
  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);
}
```

Hans Boehm of Rice University noted that using the sum of the character codes as a hash function is a poor choice. By the definition of "word" given in the

program, there are only 52 distinct character codes. So, for example, all words of length five get hashed into a range of only $5 * 52 = 260$ hash codes and words of length ten get hashed into a range of $10 * 52 = 520$. Thus, most of the hash table is empty and collisions are likely, which explains in part the large number of calls to `strcmp`. While I knew about the potentially poor performance of the hash function, I didn't change it because `common` seemed to perform adequately.

I measured the lengths of the hash chains using the test file described in the paper as input. In the following table, the right-hand column is the chain length and the left column is the number of chains of that length.

1	16
3	15
5	14
12	13
13	12
21	11
22	10
37	9
33	8
58	7
55	6
83	5
94	4
148	3
176	2
335	1
3000	0

The 3000 empty slots and long chains confirm Boehm's predictions.

Boehm suggested shifting the sum left one bit after each addition, e.g.,

```
h = 0;
s = buf;
for (len = 0; *s; len++)
    h = (h<<1) + *s++;
```

Using this hash function gives a better distribution for the test input, but there are still many empty slots:

2	10
5	9
8	8
18	7
45	6
98	5
189	4
213	3
455	2
535	1
2432	0

Finally, Joe Warren of Rice suggested mapping the character codes into random numbers and summing the random numbers. The hash function is

```
h = 0;
s = buf;
for (len = 0; *s; len++)
    h += scatter[*s++];
```

where `scatter` is initialized with the first 128 values returned by the C library function `random`. Boehm tested this function with a 4K table on a dictionary and found only 13 empty slots. This is very close to the expected value, which Boehm computed as 10.4 for the given dictionary, a 4K hash table, and assuming a uniform distribution of hash values. Using this function on the test input for `common` gave the following distribution.

1	7
3	6
18	5
62	4
242	3
769	2
1522	1
1479	0

This version of `common` (including `printwords` above) runs 8 to 9 percent faster than the published version.