**1. An example of backtracking.** Given a list of $m$-letter words and another list of $n$-letter words, we find all $m \times n$ matrices whose rows and columns are all listed. This program was written a week or so after I wrote BACK-MXN-WORDS, because I realized that I ought to try a scheme that fills in the cells of the matrix one by one. (That program fills entire columns at each level.)

I'm thinking $m = 5$ and $n = 6$ as an interesting case to try in *TAOCP*, but of course the problem makes sense in general.

The word list files are named on the command line. You can also restrict the list length to, say, at most 500 words, by appending '`:500`' to the file name.

```
#define maxm 7  /* largest permissible value of m */
#define maxn 10  /* largest permissible value of n */
#define maxmwds 30000  /* largest permissible number of m-letter words */
#define maxtriesize 1000000  /* largest permissible number of prefixes */
#define o mems++
#define oo mems += 2
#define bufsize maxm + maxn

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  unsigned long long mems;  /* memory references */
  unsigned long long thresh = 10000000000;  /* reporting time */
  typedef struct {
    int c;  /* a character (the other seven bytes are zero) */
    int link;  /* a pointer */
  } trielt;  /* a "tri element" */
  int maxmm = maxmwds, maxnn = maxtriesize;
  char mword[maxmwds][maxm + 1];
  int trie[maxtriesize][27];
  int trieptr;  /* this many nodes in the trie of n-letter words */
  trielt tri[maxtriesize];
  int triptr;  /* this many elements in the tri of m-letter words */
  char buf[bufsize];
  unsigned int count;  /* this many solutions found */
  FILE *mfile, *nfile;
  int a[maxn + 1][maxn + 1];
  int x[maxm * maxn + 1];
  long long profile[maxm * maxn + 2];
  ⟨Subroutines 8⟩;
  main(int argc, char *argv[])
  {
    register int i, j, k, l, m, n, p, q, mm, nn, xl, li, lj, lnk, chr;
    register char *w;
    ⟨Process the command line 3⟩;
    ⟨Input the m-words and make the tri 4⟩;
    ⟨Input the n-words and make the trie 5⟩;
    fprintf(stderr, "(%llu mems to initialize the data structures)\n", mems);
    ⟨Backtrack thru all solutions 9⟩;
    fprintf(stderr, "Altogether %u solutions (%llu mems).\n", count, mems);
    ⟨Print the profile 2⟩;
  }
```

**2.** ⟨ Print the profile 2 ⟩ ≡

```
fprintf(stderr, "Profile:␣␣␣␣␣␣␣␣␣␣␣1\n");
for (k = 2; k ≤ m * n + 1; k++) fprintf(stderr, "%19lld\n", profile[k]);
```

This code is used in section 1.

**3.** ⟨ Process the command line 3 ⟩ ≡

```
if (argc ≠ 3) {
  fprintf(stderr, "Usage:␣%s␣mwords[:mm]␣nwords[:nn]\n", argv[0]);
  exit(−1);
}
w = strchr(argv[1], ':');
if (w) {    /* colon in filename */
  if (sscanf(w + 1, "%d", &maxmm) ≠ 1) {
    fprintf(stderr, "I␣can't␣parse␣the␣m-file␣spec␣`%s'!\n", argv[1]);
    exit(−20);
  }
  *w = 0;
}
if (¬(mfile = fopen(argv[1], "r"))) {
  fprintf(stderr, "I␣can't␣open␣file␣`%s'␣for␣reading␣m-words!\n", argv[1]);
  exit(−2);
}
w = strchr(argv[2], ':');
if (w) {    /* colon in filename */
  if (sscanf(w + 1, "%d", &maxnn) ≠ 1) {
    fprintf(stderr, "I␣can't␣parse␣the␣n-file␣spec␣`%s'!\n", argv[1]);
    exit(−22);
  }
  *w = 0;
}
if (¬(nfile = fopen(argv[2], "r"))) {
  fprintf(stderr, "I␣can't␣open␣file␣`%s'␣for␣reading␣n-words!\n", argv[2]);
  exit(−3);
}
```

This code is used in section 1.

**4.** ⟨ Input the $m$-words and make the tri 4 ⟩ ≡

```
m = mm = 0;
while (1) {
  if (mm ≡ maxmm) break;
  if (¬fgets(buf, bufsize, mfile)) break;
  for (k = 0; o, buf[k] ≥ 'a' ∧ buf[k] ≤ 'z'; k++) o, mword[mm][k] = buf[k];
  if (buf[k] ≠ '\n') {
    fprintf(stderr, "Illegal␣m-word:␣%s", buf);
    exit(−10);
  }
  if (m ≡ 0) {
    m = k;
    if (m > maxm) {
      fprintf(stderr, "Sorry,␣m␣should␣be␣at␣most␣%d!\n", maxm);
      exit(−16);
    }
  } else if (k ≠ m) {
    fprintf(stderr, "The␣m-file␣has␣words␣of␣lengths␣%d␣and␣%d!\n", m, k);
    exit(−4);
  }
  mm++;
}
⟨ Build the tri 7 ⟩;
fprintf(stderr, "OK,␣I've␣successfully␣read␣%d␣words␣of␣length␣m=%d.\n", mm, m);
fprintf(stderr, "(The␣tri␣has␣%d␣elements.)\n", triptr);
```

This code is used in section 1.

**5.** For simplicity, I make a sparse trie with 27 branches at every node. An $n$-letter word $w_1 \ldots w_n$ leads to entries $\mathit{trie}[p_{k-1}][[w_k] = p_k$ for $1 \le k \le n$, where $p_0 = 0$ and $p_k > 0$. Here $1 \le w_k \le 26$; I am reserving slot 0 for later enhancements.

Mems are counted as if $\mathit{trie}[x][y]$ is $\mathit{array}[27 * x + y]$. (I mean, '$\mathit{trie}[x]$' is not a pointer that must be fetched, it's a pointer that the program can compute without fetching.)

```
#define trunc(c) ((c) & #1f)  /* convert 'a' to 1, ..., 'z' to 26 */

⟨Input the n-words and make the trie 5⟩ ≡
  n = nn = 0, trieptr = 1;
  while (1) {
    if (nn ≡ maxnn) break;
    if (¬fgets(buf, bufsize, nfile)) break;
    for (k = p = 0; o, buf[k] ≥ 'a' ∧ buf[k] ≤ 'z'; k++, p = q) {
      o, q = trie[p][trunc(buf[k])];
      if (q ≡ 0) break;
    }
    for (j = k; o, buf[j] ≥ 'a' ∧ buf[j] ≤ 'z'; j++) {
      if (j < n − 1 ∨ n ≡ 0) {
        if (trieptr ≡ maxtriesize) {
          fprintf(stderr, "Overflow␣(maxtriesize=%d)!\n", maxtriesize);
          exit(−66);
        }
        o, trie[p][trunc(buf[j])] = trieptr;
        p = trieptr++;
      }
    }
    if (buf[j] ≠ '\n') {
      fprintf(stderr, "Illegal␣n-word:␣%s", buf);
      exit(−11);
    }
    ⟨Check the length of the new line 6⟩;
    o, trie[p][trunc(buf[n − 1])] = nn + 1;  /* remember index of the word */
    mems −= 3;  /* we knew trie[p] when p = 0 and when q = 0; buf[j] when j = k */
    nn++;
  }
  fprintf(stderr, "Plus␣%d␣words␣of␣length␣n=%d.\n", nn, n);
  fprintf(stderr, "(The␣trie␣has␣%d␣nodes.)\n", trieptr);
```

This code is used in section 1.

**6.** ⟨ Check the length of the new line 6 ⟩ ≡

```
if (n ≡ 0) {
  n = j;
  if (n > maxn) {
    fprintf(stderr, "Sorry, n should be at most %d!\n", maxn);
    exit(−17);
  }
  p−−, trieptr −−;    /* we allocated an unnecessary node, since n wasn't known */
} else {
  if (n ≠ j) {
    fprintf(stderr, "The n-file has words of lengths %d and %d!\n", n, j);
    exit(−5);
  }
  if (k ≡ n) {
    buf[j] = 0;
    fprintf(stderr, "The n-file has the duplicate word '%s'!\n", buf);
    exit(−6);
  }
}
```

This code is used in section 5.

**7.** In this program I build what's called here (only) a "tri," by which I mean a trie that has been compressed into the following simple format: When node $k$ has $c$ children, the **trielt** entries $tri[k]$, $tri[k+1]$, $\ldots$, $tri[k+c-1]$ will contain the next character and a pointer to the relevant child node. The following entry, $tri[k+c]$, will be zero. (More precisely, its *link* part will be zero.)

It's easiest to build a normal trie first, and to compress it afterwards. So the following code—which is actually performed *before* the $n$-letter words are input—uses the *trie* array to do this.

⟨ Build the tri 7 ⟩ ≡

```
for (i = 0, trieptr = 1; i < mm; i++) {
  for (o, k = p = 0; k < m; k++, p = q) {
    o, q = trie[p][trunc(mword[i][k])];
    if (q ≡ 0) break;
  }
  for (j = k; j < m; j++) {
    if (j < m − 1) {
      if (trieptr ≡ maxtriesize) {
        fprintf(stderr, "Overflow (maxtriesize=%d)!\n", maxtriesize);
        exit(−67);
      }
      o, trie[p][trunc(mword[i][j])] = trieptr;
      p = trieptr++;
    }
  }
  o, trie[p][trunc(mword[i][m − 1])] = i + 1;    /* remember the word */
}
compress(0, m);
```

This code is used in section 4.

**8.** With a small change (namely to cache the value of a compressed node) this program would actually compress a trie that has overlapping subtries. But our trie doesn't have that property, so I don't worry about it here.

⟨ Subroutines 8 ⟩ ≡

```
int compress(int p, int l)
{
  register int a, c, k;
  oo, oo;    /* subroutine call overhead */
  if (l ≡ 0) return p;    /* prefix has its maximum length */
  for (c = 0, k = 1; k < 27; k++)
    if (o, trie[p][k]) c++;
  a = triptr;
  triptr += c + 1;
  if (triptr ≥ maxtriesize) {
    fprintf(stderr, "Tri␣overflow␣(maxtriesize=%d)!\n", maxtriesize);
    exit(−67);
  }
  for (c = 0, k = 1; k < 27; k++)
    if (o, trie[p][k]) {
      o, tri[a + c].link = compress(trie[p][k], l − 1);
      tri[a + c++].c = k;
      o, trie[p][k] = 0;    /* clear the trie for the next user */
    }
  return a;
}
```

This code is used in section 1.

**9.** Here I follow Algorithm 7.2.2B.

⟨ Backtrack thru all solutions 9 ⟩ ≡

```
b1: l = 1;
  for (k = 0; k < m; k++) o, a[k][0] = 0;   /* root of trie at left of each row */
b2: profile[l]++;
  ⟨ Report the current state, if mems ≥ thresh 11 ⟩;
  if (l > m * n) ⟨ Print a solution and goto b5 10 ⟩;
  li = (l - 1) % m;
  lj = ((l - 1)/m) + 1;   /* at this level we work on row li and column lj */
  if (li) xl = lnk;
  else xl = 0;   /* root of tri at top of each column */
  o, lnk = tri[xl].link;
b3: chr = tri[xl].c;   /* no mem cost, this word has already been fetched */
  oo, q = trie[a[li][lj - 1]][chr];
  if (¬q) goto b4;
  o, x[l] = xl;
  o, a[li][lj] = q;
  l++;
  goto b2;
b4: o, lnk = tri[++xl].link;
  if (lnk) goto b3;
b5: l--;
  if (l) {
    o, xl = x[l];
    li = (l - 1) % m;
    lj = ((l - 1)/m) + 1;
    goto b4;
  }
```

This code is used in section 1.

**10.** ⟨ Print a solution and **goto** *b5* 10 ⟩ ≡

```
  {
    count++; printf("%d:", count);
    for (k = 1; k ≤ n; k++) printf("␣%s", mword[tri[x[m * k]].link - 1]);
    for (p = 0, k = 1; k ≤ n; k++)
      if (tri[x[m * k]].link > p) p = tri[x[m * k]].link;
    for (q = 0, j = 1; j ≤ m; j++)
      if (a[j - 1][n] > q) q = a[j - 1][n];
    printf("␣(%06d,%06d;␣sum␣%07d,␣prod␣%012d)\n", p, q, p + q, p * q);
    goto b5;
  }
```

This code is used in section 9.

**11.** ⟨ Report the current state, if *mems* ≥ *thresh* 11 ⟩ ≡

```
  if (mems ≥ thresh) {
    thresh += 10000000000;
    fprintf(stderr, "After␣%lld␣mems:", mems);
    for (k = 2; k ≤ l; k++) fprintf(stderr, "␣%lld", profile[k]);
    fprintf(stderr, "\n");
  }
```

This code is used in section 9.

## 12. Index.

# BACK-MXN-WORDS-MXN