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 improves on BACK-MXN-WORDS by using a more sophisticated data structure for the m-letter words, significantly decreasing the number of candidates tested (I hope).

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 20000  /* largest permissible number of m-letter words */
#define maxtriesize 1000000  /* largest permissible number of n-letter prefixes */
#define o mems++
#define oo mems += 2
#define ooo mems += 3
#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 */
  int maxmm = maxmwds, maxnn = maxtriesize;
  char mword[maxmwds][maxm + 1];
  int mlink[maxmwds + 1][maxm];
  int head[maxm][26], size[maxm][26];
  int trie[maxtriesize][27];
  int trieptr;
  char buf[bufsize];
  unsigned int count;  /* this many solutions found */
  FILE *mfile, *nfile;
  int a[maxn + 1][maxn + 1];
  int x[maxn + 1], y[maxn + 1], z[maxn + 1];
  long long profile[maxn + 2], weight[maxn + 2];
  main(int argc, char *argv[])
  {
    register int i, j, k, l, m, n, p, q, mm, nn, t, xl, yl, zl;
    register char *w;
    ⟨Process the command line 3⟩;
    ⟨Input the m-words 4⟩;
    ⟨Input the n-words and make the trie 6⟩;
    fprintf(stderr, "(%llu mems to initialize the data structures)\n", mems);
    ⟨Backtrack thru all solutions 8⟩;
    fprintf(stderr, "Altogether %u solutions (%llu mems).\n", count, mems);
    ⟨Print the profile 2⟩;
  }
```

2. ⟨Print the profile 2⟩ ≡

```
  fprintf(stderr, "Profile:          1%9.1f\n", (double) weight[1]);
  for (k = 2; k <= n + 1; k++)
    fprintf(stderr, "%19lld%9.1f\n", profile[k], profile[k] ? weight[k]/(double) profile[k] : 0.0);
```

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 4 ⟩ ≡

```
m = mm = 0;
while (1) {
  if (mm ≡ maxmm) break;
  if (¬fgets(buf, bufsize, mfile)) break;
  mm++;
  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);
  }
  ⟨ Build sublists for each character position 5 ⟩;
}
fprintf(stderr, "OK,␣I've␣successfully␣read␣%d␣words␣of␣length␣m=%d.\n", mm, m);
```

This code is used in section 1.

5. For $0 \le k < m$ we make 26 lists, one for each word that has a given letter $j +$ 'a' in the $(k+1)$st position. The first such word is *mword* number $head[k][j]$; the next such word following word x is number $mlink[x][k]$; these links terminate with zero.

The least significant bits of the characters in *buf* could have been packed into a register, so we don't charge any mems for "fetching" them here.

⟨ Build sublists for each character position 5 ⟩ ≡

```
for (k = 0; k < m; k++) {
  j = trunc(buf[k]) − 1;      /* no mem charged, see above */
  o, p = head[k][j];      /* get the head of the j-list for position k */
  o, head[k][j] = mm;      /* insert word mm into this list */
  o, mlink[mm][k] = p;
  oo, size[k][j]++;      /* and remember the current list length */
}
```

This code is used in section 4.

6. 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 $trie[p_{k-1}][[w_k] = p_k$ for $1 \le k \le n$, where $p_0 = 0$ and $p_k > 0$. Here $1 \le w_k \le 26$.

Slot 0 of $trie[p]$ contains a bit pattern that will be helpful later: If the other slots $j_1 + 1, \ldots, j_r + 1$ have nonzero entries, we put the "signature" $\sum_{i=1} r(2^{j_i}$ into $trie[p][0]$.

Mems of statically allocated arrays like *trie* are counted as if $trie[x][y]$ is $array[27*x+y]$. (I mean, '$trie[x]$' 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 6 ⟩ ≡

```
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 (trieptr ≡ maxtriesize) {
      fprintf(stderr, "Overflow␣(maxtriesize=%d)!\n", maxtriesize);
      exit(−66);
    }
    i = trunc(buf[j]);
    oo, trie[p][0] += (1 ≪ (i − 1));
    if (j < n − 1 ∨ n ≡ 0) {
      o, trie[p][i] = trieptr;
      p = trieptr++;
    }
  }
  if (buf[j] ≠ '\n') {
    fprintf(stderr, "Illegal␣n-word:␣%s", buf);
    exit(−11);
  }
  ⟨ Check the length of the new line 7 ⟩;
  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.

7. ⟨ Check the length of the new line 7 ⟩ ≡

```
if (n ≡ 0) {
  n = j;
  p−−, trieptr −−;    /* we allocated an unnecessary node, since n wasn't known */
  if (n > maxn) {
    fprintf(stderr, "Sorry,␣n␣should␣be␣at␣most␣%d!\n", maxn);
    exit(−17);
  }
} 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 6.

8. Here I follow Algorithm 7.2.2B.

⟨ Backtrack thru all solutions 8 ⟩ ≡

```
b1: l = 1;
  for (k = 1; k ≤ m; k++) o, a[0][k] = 0;
b2: profile[l]++;
  ⟨ Report the current state, if mems ≥ thresh 11 ⟩;
  if (l > n) ⟨ Print a solution and goto b5 10 ⟩;
  ⟨ Choose a good position zl and its relevant signature yl 9 ⟩;
  i = 0;
next_i: while (((1 ≪ i) & yl) ≡ 0) i++;
  o, xl = head[zl][i];
  if (xl ≡ 0) goto new_i;
b3: o, w = mword[xl];    /* think of w's chars all in a register now, memwise */
  for (k = 1; k ≤ m; k++) {
    oo, q = trie[a[l − 1][k]][trunc(w[k − 1])];
    if (¬q) goto b4; else o, a[l][k] = q;
  }
  ooo, x[l] = xl, y[l] = yl, z[l] = zl, l++;
  goto b2;
b4: o, xl = mlink[xl][zl];
  if (xl) goto b3;    /* move to the next m-word on sublist i */
new_i: if ((1 ≪ (++i)) ≤ yl) goto next_i;
b5: l−−;
  if (l) {
    ooo, xl = x[l], yl = y[l], zl = z[l];
    o, i = mword[xl][zl] − 'a';    /* this is the subtlest part */
    goto b4;
  }
```

This code is used in section 1.

9. The kth letter of the next m-word must belong to the subset s_k that is specified in slot 0 of $trie[a[l-1][k]]$. We set zl to a $k-1$ that minimizes the corresponding sum of sublist sizes, and let yl be the corresponding subset.

⟨ Choose a good position *zl* and its relevant signature *yl* 9 ⟩ ≡

```
for (k = 1, p = maxmm + 1; k ≤ m; k++) {
  for (oo, t = trie[a[l − 1][k]][0], q = 0, i = 0; (1 ≪ i) ≤ t; i++)
    if ((1 ≪ i) & t) o, q += size[k − 1][i];
  if (q < p) p = q, zl = k − 1, yl = t;
}
weight[l] += p;    /* record the size of subdomain (for statistics only) */
```

This code is used in section 8.

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

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

This code is used in section 8.

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 8.

12. Index.

BACK-MXN-WORDS-NEW