```
/* Cryptoquote solution without trying all 403291461126605635584000000
   possibilities (with any luck and a good /usr/dict/words)

   Copyright (C) 1997 Nathan I. Laredo

   This program is modifiable/redistributable under the terms
   of the GNU General Public Licence.

   You should have received a copy of the GNU General Public License
   along with this program; if not, write to the Free Software
   Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
   Send your comments and all your spare pocket change to
   the address found in "whois tinycode.com"

   use in web pages:
   on stdin:
   "j=blahblahbalh"  solve jumble
   "anythingelse=blahblah blah" solve cryptoquote

   cryptoquote is a registered trademark of someone.
 */
/* Location of an ascii list of words separated by newline */
#define DICTIONARY_FILE "fatdict"
/* Maximum number of word matches per pass */
#define MAX_MATCHES 30000
#define MAXWORDLENGTH 64
#define NO_STATUS

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>


char *xlate[MAX_MATCHES], *xlate2[MAX_MATCHES], **windex, **in_word;
/* initial translate matrix supplied by user */
char xlc[32] __attribute__ ((aligned(32)));
/* hold starts of various length words in the dictionary */
int windexlen[MAXWORDLENGTH];
int windexlen_end[MAXWORDLENGTH];
int in, dict_size = 0;

void do_exit(i)
int i;
{
    printf("</CENTER>%d words in dict</BODY></HTML>\n", dict_size);
    exit(i);
}
int count_words(source, nchars)
char *source;
int nchars;
{
    int i, count, state;

    for (i = state = count = 0; i < nchars; i++)
        if (isalpha(source[i]))
            state = 1;
        else if (state)
            (state = 0, count++);

    return count;
}

int make_wordlist(index, source, nchars)
char **index, *source;
int nchars;
{
    int i, count, state;

    for (i = state = count = 0; i < nchars; i++)
        if (isalpha(source[i])) {
            source[i] = toupper(source[i]);
            if (!state)
```

```
                (state = 1, index[count++] = &source[i]);
        } else if (state)
            (source[i] = '\0', state = 0);
    return count;
}

int read_dict()
{
    FILE *f;
    long fsize, count;
    char *words;

    if ((f = fopen(DICTIONARY_FILE, "r")) == NULL) {
        perror(DICTIONARY_FILE);
        exit(1);
    }
    if (fseek(f, 0L, SEEK_END)) {
        perror(DICTIONARY_FILE);
        exit(1);
    }
    fsize = ftell(f);
    if (fseek(f, 0L, SEEK_SET)) {
        perror(DICTIONARY_FILE);
        exit(1);
    }
    /* save room to null terminate last value if no newline */
    if ((words = malloc(fsize + 1)) == NULL) {
        fprintf(stderr, "malloc failed for %ld bytes\n", fsize);
        do_exit(1);
    }
    if ((count = fread(words, 1, fsize, f)) < fsize) {
        fprintf(stderr, "fread error: %ld of %ld read\n", fsize, count);
        do_exit(1);
    }
    if (fclose(f))
        perror(DICTIONARY_FILE);

    count = count_words(words, fsize);

    if ((windex = malloc(count * sizeof(char *))) == NULL) {
        fprintf(stderr, "malloc failed for %ld bytes\n",
                count * sizeof(char *));
        do_exit(1);
    }
    return make_wordlist(windex, words, fsize);
}

/* for qsort: sort longest to shortest strings */
int strlencmp(a, b)
char **a, **b;
{
    return strlen(*b) - strlen(*a);
}

int strordcmp(a, b)
char **a, **b;
{
        return strcmp(*a, *b);
}

int findchar(s, c)
char *s;
int c;
{
    int i;
    for (i = 0; i < 26; i++)
        if (s[i] == c)
            return 1;
    return 0;
}

#ifndef NO_STATUS
static unsigned long int num_compares = 0;
```

```
static int mst = 0, wst = 0, mnst = 0;
#endif

/* return zero if pattern of characters matches */
int pcompare(crypted, plain, xl, l)
char *crypted, *plain, *xl;
int l;
{
    register int i;
#ifndef NO_STATUS
    num_compares++;
    /* print status every 65536 compares */
    if (!(num_compares & 0xFFFF)) {
        i = (num_compares & 0x30000) >> 16;
        printf("[%03d/%06d/%06d]%c\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b",
               wst, mnst, mst,
               i == 0 ? '|' : i == 1 ? '/' : i == 2 ? '-' : '\\');
        fflush(stdout);
    }
#endif

    /* string length already compared */
    for (i = 0; i < l; i++) {
        if (xl[crypted[i] - 'A'] == '_' && !findchar(xl, plain[i]))
            xl[crypted[i] - 'A'] = plain[i];
        if (xl[crypted[i] - 'A'] != plain[i])
            return 1;
    }
    return 0;
}

/* try to narrow down match list to a smaller list
 * compares all matches with specified word for new list of matches
 * runs through mB matches in xlateB, creating new list xlateA with
 * mA matches.   Return number of new matches found.
 * For first pass, xlateB = NULL and supplied translation (if any)
 * is used.
 */
int narrow_matches(xlateA, xlateB, mB, word)
char **xlateA, **xlateB;
int mB, word;
{
    int i, k, mA, start, len, end;

    len = strlen(in_word[word]);
    if ((start = windexlen[len]) < 0)
        return 0;
    end = windexlen_end[len];
#ifndef NO_STATUS
    mnst = 0;
    wst = word;
#endif
    for (k = mA = 0; k < mB && mA < MAX_MATCHES; k++) {
#ifndef NO_STATUS
        mst = mB - k;
#endif
        for (i = start; i < end && mA < MAX_MATCHES; i++) {
            if (xlateB != NULL)
                memcpy(xlateA[mA], xlateB[k], 26);
            else
                memcpy(xlateA[mA], xlc, 26);
            if (!pcompare(in_word[word], windex[i], xlateA[mA], len))
#ifndef NO_STATUS
                mnst =
#endif
                    mA++;
        }
    }
    return mA;

}
void frequencycount(s, out)
char *s, *out;
```

```
{
    int i;

    memcpy(out, "@@@@@@@@@@@@@@@@@@@@@@@@@@", 26);
    for (i = 0; i < strlen(s); i++)
        if (isalpha(s[i]))
            out[toupper(s[i]) - 'A']++;
}

int  freqcmp(in, out)
char *in, *out;
{
    int i;
    for  (i = 0; i < 26; i++)
        if (out[i] > in[i])
           return 1;
    return 0;
}

int main(argc, argv)
int argc;
char **argv;
{
    char instring[32767], oldstr[32767], *tmp, dowhat;
    int i, j, k, bytesread, count, use, start, end, len;
    int m1 = 0, m2 = 0, roothit = -1;

    memset(xlc, '_', 26);
    xlc[26] = '\0';
    for (j = 0, i = 1; i < argc && j < 32250; i++) {
        if (strlen(argv[i]) == 3 && argv[i][1] == '=' && isalpha(*argv[i]))
            xlc[toupper(*argv[i]) - 'A'] = toupper(argv[i][2]);
        snprintf(&instring[j], 510, "%s ", argv[i]);
        j = strlen(instring);
    }
    instring[j] = '\0';

    printf("Content-type: text/html\n\n"
           "<HTML>\n<HEAD><TITLE>Puzzle Solver Thinger... </TITLE></HEAD>\n"
           "<BODY BGCOLOR=\"#000000\" TEXT=\"#F5DEB3\">"
           "<H1>Puzzle Solver Thinger...</H1>\n"
           "<FORM METHOD=\"GET\" ACTION=\"/cgi-bin/decrypto.cgi\"><BR>\n"
           "<INPUT TYPE=\"TEXT\" SIZE=\"32\" NAME=\"j\"> "
           "Jumble solver thinger</FORM>\n"
           "<FORM METHOD=\"GET\" ACTION=\"/cgi-bin/decrypto.cgi\"><BR>\n"
           "<INPUT TYPE=\"TEXT\" SIZE=\"32\" NAME=\"q\"> "
           "Cryptoquote solver</FORM><CENTER>\n");

    if (strlen(instring) < 5)
        do_exit(0);

    /* An ugly method to allocate a string, but simplifies other stuff */

    if ((tmp = malloc(32 * MAX_MATCHES * 2)) == NULL) {
        fprintf(stderr, "malloc failed for %d bytes\n", 32 * MAX_MATCHES * 2);
        do_exit(1);
    }
    for (i = 0; i < MAX_MATCHES; i++) {
        xlate[i] = tmp;
        tmp += 32;
    }
    for (i = 0; i < MAX_MATCHES; i++) {
        xlate2[i] = tmp;
        tmp += 32;
    }
    /* setup initial translation and modify by command line args if any */
    /* almost every clue provided will exponentially decrease runtime */
    dict_size = read_dict();
    qsort(windex, dict_size, sizeof(char *), strordcmp);
    /* sort + uniq the output list */
    for (j = 1, i = 1; i < dict_size; i++) {
        if (strcmp(windex[i], windex[i - 1]))
                windex[j++] = windex[i];
```

```
    }
    dict_size = j;
    /* sort uniq dict by word length */
    qsort(windex, dict_size, sizeof(char *), strlencmp);

    /* index dictionary by word length */
    for (i = 0; i < MAXWORDLENGTH; i++)
        windexlen[i] = windexlen_end[i] = -1;

    for (i = j = 0; i < dict_size; i++) {
        k = strlen(windex[i]);
        if (j != k) {
            if (j)              /* this is buggy */
                windexlen_end[j] = i;
            windexlen[j = k] = i;
        }
    }
    dowhat = instring[0];
    tmp = strchr(instring,'=');
    if (tmp)
        strcpy(instring, (tmp + 1));
    bytesread = strlen(instring);

    strcpy(oldstr, instring);
    count = count_words(instring, bytesread);
    if ((in_word = malloc(count * sizeof(char *))) == NULL) {
        fprintf(stderr, "malloc failed for %d bytes\n",
                count * sizeof(char *));
        do_exit(1);
    }
    in = make_wordlist(in_word, instring, bytesread);
    if (dowhat != 'j')
        printf("Cryptoquote Solver<p>%d words read, processing...<br>\n", in);
    else
        printf("Jumble/Word Game Solver<p>\n");
    fflush(stdout);
    if (dowhat == 'j') {
        printf("<p>Solving for jumble/word game/scrabble puzzle: %s<p>\n", in_word[0
]);
        len = strlen(in_word[0]);
        start = windexlen[len];
        end = windexlen_end[3];
        frequencycount(in_word[0], xlate[0]);
        if (start < 0)
            do_exit(1);
        for (i = start, m1 = 0; i < end && m1 < MAX_MATCHES; i++) {
            frequencycount(windex[i], xlate[1]);
            if (!freqcmp(xlate[0], xlate[1]))
                printf("%s\n", windex[i]);
        }
        printf("<p>\n\n");
        do_exit(0);
    }

    /*
     * Here's the algorithm:
     * First, find longest possible match of input words.
     * Take those matches, and apply to all remaining words from
     * longest word (roothit) to shortest word (length > 1)
     * and see what fits list of matches from previous compare.
     */

    /* re-order word list from longest to shortest using qsort */
    qsort(in_word, in, sizeof(char *), strlencmp);

    /* Stage 1:  find matches for longest possible word */
    use = 1; /* use = 1 for xlate, 2 for xlate2 (for output at end) */
    for (m1 = j = 0; roothit < 0 && j < in; j++)
        if((m1 = narrow_matches(xlate, NULL, 1, j))>0)
            roothit = j;

    if (!m1) {
        printf("... Match Failed.  dict_size=%d words.<p>\n", dict_size);
```

```
        do_exit(1);
    }

    /* find index of last word with more than one character */
    for (j = in - 1; j > roothit && strlen(in_word[j]) < 2; j--);

    /* Stage 2:  attack all words from shortest to longest */
    if (in > 1) {
        for (i = roothit + 1; i <= j; i++) {
            if (use == 1) {
                m2 = narrow_matches(xlate2, xlate, m1, i);
                /* ignore words with too many or no matches */
                if (m2)
                    use = 2;
            } else {
                m1 = narrow_matches(xlate, xlate2, m2, i);
                /* ignore words with too many or no matches */
                if (m1)
                    use = 1;
            }
        }

    }
    if (use == 2)
        count = m2;
    else
        count = m1;

    if (count == 0) {           /* this should NEVER happen */
        printf("<p>... Sanity check failed!\n"
               "email cryptoquote to laredo@gnu.ai.mit.edu<p>\n");
        do_exit(0);
    }
    printf("... Found %d possible solution%s<p>\n", count,
           count < 2 ? ":\n" : "s:\n");

    for (j = 0; j < count; j++) {
        if (use == 1)
            tmp = xlate[j];
        else
            tmp = xlate2[j];
        for (i = 0; k = oldstr[i]; i++) {
            if (isalpha(k)) {
                if (isupper(k))
                    putchar(tmp[k - 'A']);
                else
                    putchar(tolower(tmp[k - 'a']));
            } else putchar(' ');
        }
        printf("<br>\n");
    }
    do_exit(0);
}
```