1* Intro. This program determines whether a given free tree, S, is isomorphic to a subtree of another free tree, T, using an algorithm published by David W. Matula [*Annals of Discrete Mathematics* **2** (1978), 91–106]. His algorithm is quite efficient; indeed, it runs even faster than he thought it did! If S has m nodes and T has n nodes, the running time is at worst proportional to mn times the square root of the maximum inner-degree of any node in S, where the inner degree of a node is the number of its nonleaf neighbors.

This version of the program tries every pair of free trees S and T, where S has m nodes and T has n nodes. (I hacked it by modifying the tree-generation routine of GRACEFUL-TREES.)

2*. The program is instrumented to record the number of mems, namely the number of times it accesses an octabyte of memory. (Most of the memory accesses are actually to tetrabytes (**int**s), because this program rarely deals with two tetrabytes that are known to be part of the same octabyte.)

```
#define maxn 16 /* could be increased if desired, but probably not by much */
#define maxtrees 32768 /* there are 32508 free trees of size ≤ 16 */
#define maxmtrees 128 /* there are 115 oriented trees of size 7 */
#define maxindex maxtrees + maxmtrees * maxmtrees
#define o mems++ /* count one mem */
#define oo mems += 2 /* count two mems */
#define ooo mems += 3 /* count three mems */
#define oooo mems += 4 /* count four mems */
#define suboverhead 10 /* mems charged per subroutine call */
#define decode(c) ((c) ≥ '0' ∧ (c) ≤ '9' ? (c) − '0' : (c) ≥ 'a' ∧ (c) ≤ 'z' ? (c) − 'a' + 10 :
        (c) ≥ 'A' ∧ (c) ≤ 'Z' ? (c) − 'A' + 36 : −1)
#define encode(p) ((p) < 10 ? (p) + '0' : (p) < 36 ? (p) − 10 + 'a' : (p) < 62 ? (p) − 36 + 'A' : '?')
#include <stdio.h>
#include <stdlib.h>
  ⟨ Type definitions 9* ⟩;
  ⟨ Global variables 4* ⟩;
#include <math.h>
  int mm, nn; /* command-line parameters */
  unsigned long long mems; /* memory references */
  unsigned long long imems; /* mems during the input phase */
  ⟨ Subroutines 5* ⟩;
  main(int argc, char *argv[])
  {
    register int d, e, g, i, j, k, m, n, p, q, r, s, v, z;
    ⟨ Process the command line 3* ⟩;
    ⟨ Build a trie for locating all m-vertex free trees 11* ⟩;
    ⟨ Build a trie for locating all n-vertex free trees 23* ⟩;
    imems = mems, mems = 0;
    ⟨ Set up S, the first m-node tree 12* ⟩;
    while (1) {
      ⟨ Set up T, the first n-node tree 24* ⟩;
      while (1) {
        startmems = mems;
        ⟨ Solve the problem 37 ⟩;
        ⟨ Record the solution 62* ⟩;
        ⟨ Change T to the next n-node tree, or break 27* ⟩;
      }
      ⟨ Change S to the next m-node tree, or break 18* ⟩;
    }
    ⟨ Sign off 65* ⟩;
  }
```

3*. ⟨ Process the command line 3* ⟩ ≡

```
if (argc ≠ 3 ∨ sscanf(argv[1], "%d", &mm) ≠ 1 ∨ sscanf(argv[2], "%d", &nn) ≠ 1) {
  fprintf(stderr, "Usage: %s m n\n", argv[0]);
  exit(−1);
}
if (mm < 3 ∨ mm > nn ∨ nn > maxn) {
  fprintf(stderr, "Sorry, I'm configured to handle only 2 < m <= n <= %d.\n", maxn);
  exit(−2);
}
m = mm, n = nn;
```

This code is used in section 2*.

4* The trie of all free trees. Let $m = \lfloor (n-1)/2 \rfloor$. According to the theory in exercise 7.2.1.6–90 of *The Art of Computer Programming*, every free tree on n vertices is either centroidal or bicentroidal: It either has a unique centroid, in which case the other vertices form an oriented forest, having no trees of size $> m$; or it has two centroids, in which case the children of each centroid form oriented forests of size m. The bicentroidal case occurs only when n is even.

We shall construct a table of all oriented forests of size $< n$, containing no tree of size $> m$. Each t-node oriented forest is represented by its canonical level sequence $c_1 \dots c_t$, where c_k is the depth of the kth node in preorder, for $1 \le k \le t$. The sequence is canonical if the substrings for sibling subtrees in each family appear in nonincreasing lexicographic order.

Say that $c_1 \dots c_t$ is *legal* if it is canonical for an oriented forest with no tree of size $> m$. For example, suppose $m = 4$. The sequence 02010101 is illegal because it isn't a level sequence. (In a level sequence we always have $c_{k+1} \le c_k + 1$.) The sequence 01110120 is illegal because it's not canonical ($0111 < 012$). The sequence 01201111 is illegal because 01111 is a tree of size 5. The sequence 01201120 is illegal because it's not canonical ($1 < 12$). The sequence 01201110 is legal, and so is 012012010. Our job is to tabulate every legal $c_1 \dots c_t$ with $t < n$.

We shall generate the legal sequences for $t = 1$, then $t = 2$, ..., then $t = n - 1$. And for fixed t, we'll generate them in decreasing lexicographic order, as in exercise 7.2.1.6–90, starting with the largest — which consists of the first t elements of the cyclic sequence $012 \dots (m{-}1)012 \dots (m{-}1)0 \dots$.

If $c_1 \dots c_t$ is legal, so is its prefix $c_1 \dots c_{t-1}$, assuming that $t > 1$. So also is its lexicographic predecessor $c_1 \dots c_{t-1}(c_t{-}1)$, assuming that $c_t > 0$. Thus the legal sequences form a trie.

We shall build a trie data structure with two arrays, called c and up. If k represents $c_1 \dots c_t$ then $c[k] = c_t$; $up[k]$ represents $c_1 \dots c_{t-1}$; and $k + 1$ represents $c_1 \dots c_{t-1}(c_t{-}1)$, if $c_t > 0$.

This program also computes the associated sequence of *parent* pointers, $p_1 \dots p_t$, by adding a third array called np, where $np[k] = p_t$ when k represents $c_1 \dots c_t$. For example, the parent pointers that correspond to 012110121010 are 012110454070.

```
⟨ Global variables 4* ⟩ ≡
  void make_sstring(int k);
  void make_tstring(int k);
  char sstring[maxn + 1] = ".", tstring[maxn + 1] = ".";
  int up[maxtrees];      /* parent in trie */
  int down[maxtrees];    /* leftmost child in trie */
  int c[maxtrees];       /* c_t coordinate in trie */
  int np[maxtrees];      /* p_t coordinate in trie */
  int ptr;       /* the first unused entry of up, c, and np */
  int cc[maxn];      /* the current level sequence */
  int pp[maxn];      /* the current parent sequence */
  int start[maxn];       /* where forests of each size begin */
```

See also sections 10, 13*, 25*, 36, 42, 50, 55, 59, and 64*.

This code is used in section 2*.

5*. When *maketrie*(n) is done, *start*$[t]$ will be the number of forests of size $< t$ that contain no tree of size exceeding $\lfloor (n-1)/2 \rfloor$, for $1 \le t \le n$.

(This program assumes that $n > 2$.)

```
⟨ Subroutines 5* ⟩ ≡
  void maketrie(int n)
  {
    register int i, j, k, l, m, q, t, cstar;
    m = (n − 1) ≫ 1;
    o, start[1] = k = 1, ptr = 2;      /* up[1] = c[1] = 0 handles the first sequence, c_1 */
    oo, cc[0] = pp[0] = −1;      /* "the level above the forest" */
    for (t = 1; t < n − 1; t++) ⟨ Generate the sequences c_1 ... c_{t+1} from the sequences c_1 ... c_t 6* ⟩;
    if (oo, start[m + 1] − start[m] > maxmtrees) {
      fprintf(stderr, "Recompile␣me␣with␣maxmtrees>=%d!\n", start[m + 1] − start[m]);
      exit(−66);
    }
    o, start[n] = ptr;
  }
```

See also sections 17*, 32, 38, and 66*.

This code is used in section 2*.

6*. At this point $k = start[t]$.

When $k > start[t]$, the computation for k usually has a lot in common with the computation for $k-1$. Therefore there's considerable potential for optimization. But I've opted for simplicity today.

```
⟨ Generate the sequences c_1 ... c_{t+1} from the sequences c_1 ... c_t 6* ⟩ ≡
  {
    for (o, start[t + 1] = ptr; k < start[t + 1]; k++) {
      for (i = t, j = k, q = −1; i; i−−, o, j = up[j]) {
        oooo, cc[i] = c[j], pp[i] = np[j];
        if (cc[i] ≡ 0 ∧ q < 0) q = i;      /* q is position of rightmost root */
      }
      ⟨ Determine c* for the sequence c_1 ... c_t 7* ⟩;
      o, down[k] = ptr;
      for (o, q = t, j = cc[t] − cstar; j ≥ 0; o, j−−, q = pp[q]) ;
      for (j = cstar; j ≥ 0; j−−) oooo, up[ptr] = k, c[ptr] = j, np[ptr] = q, q = pp[q], ptr++;
    }
  }
```

This code is used in section 5*.

7*. The goal here is to determine c^*, the largest level that can legally follow the sequence of levels $c_1 \dots c_t$.

```
⟨ Determine c* for the sequence c_1 ... c_t 7* ⟩ ≡
  if (q + m ≡ t + 1) cstar = 0;      /* the final tree already has m nodes */
  else
    for (o, l = cc[t], cstar = l + 1, j = t; l ≥ 0; o, l−−, j = pp[j]) {
      ⟨ Check canonicity at level l 8* ⟩;
    }
```

This code is used in section 6*.

8*. At this point j is maximal with $cc[j] = l$; this means that j is the ancestor of t at level l. If j has a left sibling, we decrease *cstar* if necessary so that the substring starting at j doesn't exceed the substring starting at that sibling.

```
⟨ Check canonicity at level l 8* ⟩ ≡
  if (o, cc[j − 1] ≥ l) {      /* yes, there is a left sibling */
    for (q = j − 1; o, cc[q] > l; q−−) ;      /* find where its subtree begins */
    for (i = 1; j + i ≤ t; i++)
      if (oo, cc[q + i] ≠ cc[j + i]) break;
    if (j + i > t) {
      if (o, cstar > cc[q + i]) cstar = cc[q + i];      /* retain lexicographic order */
    } else if (cc[q + i] < cc[j + i]) fprintf(stderr, "I'm␣confused!\n");
        /* previous lexicographic test failed */
  }
```

This code is used in section 7*.

9* Data structures for the trees. A **node** record is allocated for each node of a tree. It has four fields: *child* (the index of its most recent child, if any), *sib* (the index of its parent's previous child, if any), *deg* (the number of neighbors), and *arc* (the number of the arc to its parent). The *deg* and *arc* fields aren't actually used for S, but we need them for T. Reference to the *deg* and *arc* fields in the same node counts as only one mem.

```
⟨ Type definitions 9* ⟩ ≡
  typedef struct node_struct {
    int child;      /* who is my first child, if any? */
    int sib;      /* who is the next child of my parent, if any? */
    int deg;      /* how many neighbors do I have, including my parent (if any)? */
    int arc;      /* which arc corresponds to the link from me to my parent? */
  } node;
```

This code is used in section 2*.

10. ⟨ Global variables 4* ⟩ +≡

```
  node snode[maxn];      /* the m nodes of S */
  node tnode[maxn];      /* the n nodes of T */
```

11* ⟨ Build a trie for locating all m-vertex free trees 11* ⟩ ≡

```
  maketrie(m);
  for (o, k = 1; k < start[m]; k++) oooo, mup[k] = up[k], mp[k] = np[k];
  oo, mstart = start[m − 1], mstop = start[m];
  if ((m & 1) ≡ 0) oo, mshortstart = start[(m ≫ 1) − 1], mshortstop = start[m ≫ 1];
```

This code is used in section 2*.

12* As we proceed, S will be the tree rooted at 0 for which the parent of node k is $pm[k]$, for $1 \le k < m$.

```
⟨ Set up S, the first m-node tree 12* ⟩ ≡
  mphase = 0, mstep = mstart, mserial = 0;
  for (k = m − 1, j = mstep; k; k−−) oooo, pm[k] = mp[j], upm[k] = j = mup[j];
  ⟨ Convert the pm array into a tree S in snode 14* ⟩;
```

This code is used in section 2*.

13* ⟨ Global variables 4* ⟩ +≡

```
  int mup[maxtrees];      /* a version of up, for m-node trees */
  int mp[maxtrees];      /* version of np, for m-node trees */
  int pm[maxn];      /* the parents of nodes in the current S */
  int upm[maxn];      /* where we've been when setting pm */
  int mstart, mstop, mshortstart, mshortstop;      /* trie boundaries for making S */
  int mphase, mstep, mstepx, mserial;      /* controllers for the loop on S */
```

14*. Matula's routine will want the root of S to be a leaf. So we first use *tnode* to create the free tree specified by *pm*; then we move a leaf to root position of *tnode*; finally we produce the desired tree in *snode* by copying and remapping *tnode*. (I got this code from MATULA-BIG.)

```
⟨ Convert the pm array into a tree S in snode 14* ⟩ ≡
  o, tnode[0].child = tnode[0].sib = 0;
  for (k = 1; k < m; k++) {
    o, p = pm[k];
    oo, q = tnode[p].child, tnode[p].child = k;
    o, tnode[k].child = 0, tnode[k].sib = q;
  }
  ⟨ Make the root of tnode into a leaf 15* ⟩;
  ⟨ Copy and remap tnode into snode 16* ⟩;
```

This code is used in sections 12* and 18*.

15*. I thought this would be easier than it has turned out to be. Did I miss something? It's a nice little exercise in datastructurology.

Node 0 moves to node m, so that it can become a child or a sibling.

```
⟨ Make the root of tnode into a leaf 15* ⟩ ≡
  oo, r = m, p = tnode[0].child, tnode[r].child = p, tnode[r].sib = 0;
  while (o, q = tnode[p].child) {      /* make p the root, retaining its child q */
    o, k = tnode[p].sib, s = tnode[q].sib;
    o, tnode[p].sib = 0;
    o, tnode[q].sib = r;
    o, tnode[r].child = k, tnode[r].sib = s;
    r = p, p = q;
  }
  ooo, s = tnode[p].sib, tnode[p].sib = 0, tnode[p].child = r, tnode[r].child = s;      /* now p is the root */
```

This code is used in section 14*.

16*.

```
⟨ Copy and remap tnode into snode 16* ⟩ ≡
  for (gg = k = 0; k < m; k++) o, snode[k].child = snode[k].sib = 0;
  copyremap(p);
  if (gg ≠ m) {
    fprintf(stderr, "I'm␣basically␣confused!\n");
    exit(−666);
  }
  oo, snode[0].arc = snode[m].arc;
```

This code is used in section 14*.

17*. This recursion is a bit tricky, and I wonder what's the best way to explain it. (An exercise for the reader.)

```
⟨ Subroutines 5* ⟩ +≡
  int gg;      /* global counter for remapping */
  void copyremap(int r)
  {
    register int p, q;
    mems += suboverhead;
    gg++;
    o, p = tnode[r].child;
    if (¬p) return;
    o, snode[gg − 1].child = gg;      /* copy a (remapped) child pointer */
    while (1) {
      q = gg;      /* the future interior name of p */
      copyremap(p);
      o, p = tnode[p].sib;
      if (¬p) return;
      o, snode[q].sib = gg;      /* copy a (remapped) sibling pointer */
    }
  }
```

18*. ⟨ Change S to the next m-node tree, or **break** 18* ⟩ ≡

```
  mserial++;
  if (mphase) ⟨ Do a bicentroidal m-step change for S, or break 21* ⟩
  else if (++mstep < mstop) {
    for (k = m − 1, j = mstep; k; k−−) {
      ooo, pm[k] = mp[j], j = mup[j];
      if (o, j ≡ upm[k]) break;      /* we've been there and done that */
      o, upm[k] = j;
    }
  } else if (m & 1) break;
  else ⟨ Set up S, the first bicentroidal m-node tree 19* ⟩;
  ⟨ Convert the pm array into a tree S in snode 14* ⟩;
```

This code is used in section 2*.

19*. The bicentroidal trees require us to run two loops, for trees of size $m/2$.

```
⟨ Set up S, the first bicentroidal m-node tree 19* ⟩ ≡
  {
    mphase = 1, mstep = mshortstart;
    for (k = (m ≫ 1) − 1, j = mshortstart; k; k−−) oooo, pm[k] = mp[j], upm[k] = j = mup[j];
    ⟨ Set up the right half of bicentroidal S beginning at mstep 20* ⟩;
  }
```

This code is used in section 18*.

20*. ⟨ Set up the right half of bicentroidal S beginning at *mstep* 20* ⟩ ≡

```
  mstepx = mstep;
  for (k = m − 1, j = mstepx; k > (m ≫ 1); k−−) oooo, pm[k] = mp[j] + (m ≫ 1), upm[k] = j = mup[j];
```

This code is used in sections 19* and 22*.

21*. ⟨ Do a bicentroidal m-step change for S, or **break** 21* ⟩ ≡

```
{
  if (++mstepx ≡ mshortstop) ⟨ Change S to the next bicentroidal m-node tree, or break 22* ⟩
  else {
    for (k = m − 1, j = mstepx; k; k−−) {
      ooo, pm[k] = mp[j] + (m ≫ 1), j = mup[j];
      if (o, j ≡ upm[k]) break;      /* we've been there and done that */
      o, upm[k] = j;
    }
  }
}
```

This code is used in section 18*.

22*. ⟨ Change S to the next bicentroidal m-node tree, or **break** 22* ⟩ ≡

```
{
  if (++mstep ≡ mshortstop) break;
  for (k = (m ≫ 1) − 1, j = mstep; k; k−−) {
    ooo, pm[k] = mp[j], j = mup[j];
    if (o, j ≡ upm[k]) break;      /* we've been there and done that */
    o, upm[k] = j;
  }
  ⟨ Set up the right half of bicentroidal S beginning at mstep 20* ⟩;
}
```

This code is used in section 21*.

23*. ⟨ Build a trie for locating all n-vertex free trees 23* ⟩ ≡

```
maketrie(n);
oo, nstart = start[n − 1], nstop = start[n];
if ((n & 1) ≡ 0) oo, nshortstart = start[(n ≫ 1) − 1], nshortstop = start[n ≫ 1];
```

This code is used in section 2*.

24*. As we proceed, T will be the tree rooted at 0 for which the parent of node k is $pn[k]$, for $1 \le k < n$.

⟨ Set up T, the first n-node tree 24* ⟩ ≡

```
nphase = 0, nstep = nstart, nserial = 0;
for (k = n − 1, j = nstep; k; k−−) oooo, pn[k] = np[j], upn[k] = j = up[j];
⟨ Convert the pn array into a tree T in tnode 26* ⟩;
```

This code is used in section 2*.

25*. ⟨ Global variables 4* ⟩ +≡

```
int pn[maxn];      /* the parents of nodes in the current T */
int upn[maxn];     /* where we've been when setting pn */
int nstart, nstop, nshortstart, nshortstop;      /* trie boundaries for making T */
int nphase, nstep, nstepx, nserial;      /* controllers for the loop on T */
```

26*. The tree in *tnode* is fancier than the tree in *snode*, because Matula's algorithm will use its *deg* and *arc* fields.

```
⟨ Convert the pn array into a tree T in tnode 26* ⟩ ≡
  o, tnode[0].child = tnode[0].sib = 0;
  for (k = 1; k < n; k++) {
    o, head[k] = 0;
    o, p = pn[k];
    oo, q = tnode[p].child, tnode[p].child = k;
    o, tnode[k].child = 0, tnode[k].sib = q;
  }
  ⟨ Allocate the arcs 33 ⟩;
```

This code is used in sections 24* and 27*.

27*.
```
⟨ Change T to the next n-node tree, or break 27* ⟩ ≡
  nserial++;
  if (nphase) ⟨ Do a bicentroidal n-step change for T, or break 30* ⟩
  else if (++nstep < nstop) {
    for (k = n − 1, j = nstep; k; k−−) {
      ooo, pn[k] = np[j], j = up[j];
      if (o, j ≡ upn[k]) break;      /* we've been there and done that */
      o, upn[k] = j;
    }
  } else if (n & 1) break;
  else ⟨ Set up T, the first bicentroidal n-node tree 28* ⟩;
  ⟨ Convert the pn array into a tree T in tnode 26* ⟩;
```

This code is used in section 2*.

28*. The bicentroidal trees require us to run two loops, for trees of size $n/2$.

```
⟨ Set up T, the first bicentroidal n-node tree 28* ⟩ ≡
  {
    nphase = 1, nstep = nshortstart;
    for (k = (n ≫ 1) − 1, j = nshortstart; k; k−−) oooo, pn[k] = np[j], upn[k] = j = up[j];
    ⟨ Set up the right half of bicentroidal T beginning at nstep 29* ⟩;
  }
```

This code is used in section 27*.

29*.
```
⟨ Set up the right half of bicentroidal T beginning at nstep 29* ⟩ ≡
  nstepx = nstep;
  for (k = n − 1, j = nstepx; k > (n ≫ 1); k−−) oooo, pn[k] = np[j] + (n ≫ 1), upn[k] = j = up[j];
```

This code is used in sections 28* and 31*.

30*. ⟨ Do a bicentroidal n-step change for T, or **break** 30* ⟩ ≡

```
{
  if (++nstepx ≡ nshortstop) ⟨ Change T to the next bicentroidal n-node tree, or break 31* ⟩
  else {
    for (k = n − 1, j = nstepx; k; k−−) {
      ooo, pn[k] = np[j] + (n ≫ 1), j = up[j];
      if (o, j ≡ upn[k]) break;      /* we've been there and done that */
      o, upn[k] = j;
    }
  }
}
```

This code is used in section 27*.

31*. ⟨ Change T to the next bicentroidal n-node tree, or **break** 31* ⟩ ≡

```
{
  if (++nstep ≡ nshortstop) break;
  for (k = (n ≫ 1) − 1, j = nstep; k; k−−) {
    ooo, pn[k] = np[j], j = up[j];
    if (o, j ≡ upn[k]) break;      /* we've been there and done that */
    o, upn[k] = j;
  }
  ⟨ Set up the right half of bicentroidal T beginning at nstep 29* ⟩;
}
```

This code is used in section 30*.

32. The target tree T has $2(n-1)$ arcs, from each nonroot node to its parent and vice versa. The arcs from u to v are assigned consecutive integers, from 0 to $2n-3$, in lexicographic order of $(\deg(v), v, u)$. (Well, the second and third components might not be in numerical order; but all d arcs from a vertex of degree d are consecutive, beginning with the arc to the parent.)

In order to assign these numbers, we keep lists of all nodes having a given degree, using the *arc* fields temporarily to link them together.

```
⟨ Subroutines 5* ⟩ +≡
  void fixdeg(int p)
  {
    register int d, q;
    mems += suboverhead;
    for (o, d = 1, q = tnode[p].child; q; o, d++, q = tnode[q].sib) fixdeg(q);
    if (p) ooo, tnode[p].arc = head[d], tnode[p].deg = d, head[d] = p;
        /* p is not the root; it has d neighbors including its parent */
    else ooo, tnode[0].arc = head[d − 1], tnode[0].deg = d − 1, head[d − 1] = −1;
        /* root is temporarily renamed −1 */
  }
```

33. We set *thresh*[*d*] to the number of the first arc for a node of degree *d* or more.

```
⟨ Allocate the arcs 33 ⟩ ≡
  fixdeg(0);
  for (d = 1, e = 0; e < 2 * n − 2; d++) {
    o, thresh[d] = e;
    for (o, p = head[d]; p; e += d, p = q) {
      if (p < 0) p = 0;
      oo, q = tnode[p].arc, tnode[p].arc = e;
    }
  }
  for (maxdeg = d − 1, emax = e; d < m; d++) o, thresh[d] = emax;
  ⟨ Allocate the dual arcs 35 ⟩;
```

This code is used in section 26*.

34. The arc from *u* to *v* has a dual, namely the arc from *v* to *u*. (And conversely.) We've assigned numbers to the arcs that go to a parent; the other arcs are their duals.

35.
```
⟨ Allocate the dual arcs 35 ⟩ ≡
  for (p = 0; p < n; p++) {
    for (oo, e = (p ? tnode[p].arc : tnode[p].arc − 1), q = tnode[p].child; q; o, q = tnode[q].sib) {
      ooo, dual[tnode[q].arc] = ++e, dual[e] = tnode[q].arc;
      oooo, uert[dual[e]] = vert[e] = p, uert[e] = vert[dual[e]] = q;
    }
  }
```

This code is used in section 33.

36.
```
⟨ Global variables 4* ⟩ +≡
  int head[maxn];      /* heads of lists by degree */
  int maxdeg;      /* maximum degree seen */
  int thresh[maxn];      /* where the arcs from large degree nodes start */
  int vert[maxn + maxn];      /* the source vertex of each arc */
  int uert[maxn + maxn];      /* the target vertex of each arc */
  int dual[maxn + maxn];      /* the dual of each arc */
  int emax;      /* the total number of arcs */
```

37. The master control. There's a two-dimensional array called *sol* that pretty much governs the computation. The first index, p, is a node of S; the second index, e, is an arc of T. If e is the arc from u to v, consider the subtree of T that's rooted at u and includes v; we call it "subtree e." If there's no way to embed the subtree of S rooted at p to subtree e, by mapping p to v, then we'll set $sol[p][e]$ to zero. Otherwise we'll set $sol[p][e]$ to a nonzero value, with which we could deduce such an embedding if called on to do so.

The basic idea is simple, working recursively up from small subtrees to larger ones: Suppose p has r children, $q_1, \ldots, q_r$; and suppose v has $s+1$ neighbors, $u_0, \ldots, u_s$. Suppose further that we've already computed $sol[q_i][e_j]$, for $1 \le i \le r$ and $0 \le j \le s$, where e_j is the arc from v to u_j. Matula's algorithm will tell us how to compute $sol[p][dual[e_j]]$ for $0 \le j \le s$. Thus we can fill in the rows of *sol* from bottom to top; eventually $sol[1]$ will tell us if we can embed *all* of S.

Let's look closely at that crucial subproblem: How, for example, do we know if $sol[p][dual[e_0]]$ should be zero or nonzero? That subproblem means that we want to embed subtree p into the subtree below the arc from u_0 to v. And the subproblem is clearly solvable if and only if we can match up each child q_i of p with a distinct child u_j of v, in such a way that $sol[p_i][q_j]$ is nonzero. Aha, yes: It's a bipartite matching problem! And there are good algorithms for bipartite matching!

More generally, consider the subproblem in which u_j is a parent of v in T, while $u_0, \ldots, u_{j-1}, u_{j+1}, \ldots, u_s$ are children. Matula discovered that these subproblems are essentially the same, for all j between 0 and s. It's a beautiful way to save a factor of n by combining similar subproblems.

So that's what we'll do, with a recursive procedure called *solve*.

```
⟨ Solve the problem 37 ⟩ ≡
  z = solve(1);
```

This code is used in section 2*.

38. The task of *solve*, given a node p of S, is to set the values of $sol[p][e]$ for each arc e.

The base case of this recursion occurs when p is a leaf; a leaf can be embedded anywhere.

Another easy case occurs when subtree e of T has too small a degree to support any embedding.

If some descendant d of p can't be embedded, *solve* returns $-d$. Otherwise *solve* returns the number of 1s in $sol[p]$.

```
⟨ Subroutines 5* ⟩ +≡
  int solve(int p)
  {
    register int e, m, n, q, r, z;
    mems += suboverhead;
    o, q = snode[p].child;
    if (q ≡ 0) {
      for (e = 0; e < emax; e++) o, sol[p][e] = 1;
      return emax;
    }
    for (r = 0; q; o, r++, q = snode[q].sib) {
      z = solve(q);
      if (z ≤ 0) return (z ? z : −q);      /* if we can't embed a subtree, we can't embed S */
    }     /* now sol[q][e] is known for all children q of p and all arcs e */
    for (o, z = e = 0; e < thresh[r + 1]; e++) o, sol[p][e] = 0;      /* degree too small */
    for (n = r + 1; e < emax; e += n) {
      ⟨ Local variables for the HK algorithm 44 ⟩;
      while (o, e ≡ thresh[n + 1]) n++;      /* advance n to the degree of vert[e] */
      ⟨ Set up Matula's bipartite matching problem for p and e 39* ⟩;
      ⟨ Solve that problem and update sol[p][e . . e + n − 1] 52 ⟩;
    }
    return z;
  }
```

39* Bipartite matching chez Hopcroft and Karp. Now we implement the classic HK algorithm for bipartite matching, stealing most of the code from the program HOPCROFT-KARP. (The reader should consult that program for further remarks and proofs.) The children of p play the role of "boys" in that algorithm, and the arcs for neighbors of v play the role of "girls." That algorithm is slightly simplified here, because we are interested only in cases where all the boys can be matched. (There always are more girls than boys, in our case.)

In Matula's matching problem, p is a vertex of S that has children $q_1, \ldots, q_r$; e is an arc of T from $v = vert[e]$ to $u = uert[e]$, where v has $s+1$ neighbors $u_0, \ldots, u_s$. The matching problem will have $m \le r$ boys and $n = s+1$ girls.

We use a simple data structure to represent the bipartite graph: The potential partners for girl j are in a linked list beginning at $glink[j]$, linked in $next$, and terminated by a zero link. The partner at link l is stored in $tip[l]$.

```
⟨ Set up Matula's bipartite matching problem for p and e 39* ⟩ ≡
  ⟨ Initialize the tables needed for n girls 41 ⟩;
  for (o, t = m = 0, b = snode[p].child; b; o, b = snode[b].sib) ⟨ Record the potential matches for boy b 40 ⟩;
  if (m ≡ 0) goto yes_sol;      /* every boy matches every girl */
  if (m * n > record) {
    record = m * n;
    make_sstring(mserial);
    make_tstring(nserial);
    fprintf(stderr, "␣...matching␣%d␣boys␣to␣%d␣girls␣(%s,%s)\n", m, n, sstring, tstring);
  }
```

This code is used in section 38.

40. If b is matched to every girl, we needn't include him in the bipartite graph. (This situation happens rather often, for example whenever b is a leaf, so it's wise to test for it.) On the other hand, if some boy isn't matched to any girl, we know in advance that there will be no bipartite matching.

The HK algorithm uses a $mate$ table, to indicate the current mate of every boy as it constructs tentative matchings. There's also an inverse table, $imate$, for the girls. If b has no mate, $mate[b] = 0$; if g has no mate, $imate[g] = 0$. But if b is tentatively matched to g, we have $mate[b] = g$ and $imate[g] = b$.

```
⟨ Record the potential matches for boy b 40 ⟩ ≡
  {
    for (g = e; g < e + n; g++)
      if (oo, sol[b][dual[g]] ≡ 0) break;
    if (g ≡ e + n) continue;      /* boy b fits anywhere, so omit him */
    oo, m++, mate[b] = mark[b] = 0;
    for (k = t, gg = e; gg < g; gg++) oooo, tip[++t] = b, next[t] = glink[gg], glink[gg] = t;
    for (g++; g < e + n; g++)
      if (oo, sol[b][dual[g]]) oooo, tip[++t] = b, next[t] = glink[g], glink[g] = t;
    if (k ≡ t) goto no_sol;      /* boy b fits nowhere, so give up */
  }
```

This code is used in section 39*.

41. We've now created a bipartite graph with m boys, n girls, and t edges.

The HK algorithm proceeds in *rounds*, where each round finds a maximal set of so-called SAPs, which are are vertex-disjoint augmenting paths of the shortest possible length. If a round finds k such paths, it reduces the number of free boys (and free girls) by k. Eventually, after at most $2\sqrt{n}$ rounds, we reach a state where no more SAPs exist. And then we have a solution, if and only if no boys are still free (hence $n-m$ girls are still free).

Variable f in the algorithm denotes the current number of free girls. They all appear in the first f positions of any array called *queue*, which governs a breadth-first search. This array has an inverse, *iqueue*: If g is free, we have $queue[iqueue[g]] = g$.

```
⟨ Initialize the tables needed for n girls 41 ⟩ ≡
  for (g = e; g < e + n; g++) oooo, glink[g] = 0, imate[g] = 0, queue[g − e] = g, iqueue[g] = g − e;
  f = n;
```

This code is used in section 39*.

42. The key idea of the HK algorithm is to create a directed acyclic graph in which the paths from a dummy node called $\top$ to a dummy node called $\bot$ correspond one-to-one with the augmenting paths of minimum length. Each of those paths will contain *final_level* existing matches.

This dag has a representation something like our representation of the girls' choices, but even sparser: The first arc from boy i to a suitable girl is in $blink[i]$, with *tip* and *next* as before. Each girl, however, has exactly one outgoing arc in the dag, namely her *imate*. An *imate* of 0 is a link to $\bot$. The other dummy node, $\top$, has a list of free boys, beginning at *dlink*.

An array called *mark* keeps track of the level (plus 1) at which a boy has entered the dag. All marks must be zero when we begin.

The *next* and *tip* arrays must be able to accommodate $2t + m$ entries: t for the original graph, t for the edges at round 0, and m for the edges from $\top$.

```
#define maxg (2 * maxn)      /* upper limit on the number of girls */
#define maxt (maxn * maxg)      /* upper limit on the number of bipartite edges */
⟨ Global variables 4* ⟩ +≡
  int blink[maxn], glink[maxg];      /* list heads for potential partners */
  int next[maxt + maxt + maxn], tip[maxt + maxt + maxn];      /* links and suitable partners */
  int mate[maxn], imate[maxg];
  int queue[maxg];      /* girls seen during the breadth-first search */
  int iqueue[maxg];      /* inverse permutation, for the first f entries */
  int mark[maxn];      /* where boys appear in the dag */
  int marked[maxn];      /* which boys have been marked */
  int dlink;      /* head of the list of free boys in the dag */
```

43. ⟨ Build the dag of shortest augmenting paths (SAPs) 43 ⟩ ≡

```
final_level = −1, tt = t;
for (marks = l = i = 0, q = f; ; l++) {
  for (qq = q; i < qq; i++) {
    o, g = queue[i];
    for (o, k = glink[g]; k; o, k = next[k]) {
      oo, b = tip[k], pp = mark[b];
      if (pp ≡ 0) ⟨ Enter b into the dag 45 ⟩
      else if (pp ≤ l) continue;
      oooo, tip[++tt] = g, next[tt] = blink[b], blink[b] = tt;
    }
  }
  if (q ≡ qq) break;      /* nothing new on the queue for the next level */
}
```

This code is used in section 52.

44. ⟨ Local variables for the HK algorithm 44 ⟩ ≡

```
register int b, f, g, i, j, k, l, t, gg, pp, qq, tt, final_level, marks;
```

This code is used in section 38.

45. Once we know we've reached the final level, we don't allow any more boys at that level unless they're free. We also reset q to qq, so that the dag will not reach a greater level.

⟨ Enter b into the dag 45 ⟩ ≡

```
{
  if (final_level ≥ 0 ∧ (o, mate[b])) continue;
  else if (final_level < 0 ∧ (o, mate[b] ≡ 0)) final_level = l, dlink = 0, q = qq;
  ooo, mark[b] = l + 1, marked[marks++] = b, blink[b] = 0;
  if (mate[b]) oo, queue[q++] = mate[b];
  else oo, tip[++tt] = b, next[tt] = dlink, dlink = tt;
}
```

This code is used in section 43.

46. We have no SAPs if and only no free boys were found.

⟨ If there are no SAPs, **break** 46 ⟩ ≡

```
if (final_level < 0) break;
```

This code is used in section 52.

47. ⟨ Reset all marks to zero 47 ⟩ ≡

```
while (marks) oo, mark[marked[−−marks]] = 0;
```

This code is used in section 48.

48. We've just built the dag of shortest augmenting paths, by starting from dummy node $\perp$ at the bottom and proceeding breadth-first until discovering *final_level* and essentially reaching the dummy node $\top$. Now we more or less reverse the process: We start at $\top$ and proceed *depth*-first, harvesting a maximal set of vertex-disjoint augmenting paths as we go. (Any maximal set will be fine; we needn't bother to look for an especially large one.)

The dag is gradually dismantled as SAPs are removed, so that their boys and girls won't be reused. A subtle point arises here when we look at a girl g who was part of a previous SAP: In that case her mate will have been changed to a boy whose *mark* is negative. This is true even if $l = 0$ and g was previously free.

```
⟨ Find a maximal set of disjoint SAPs, and incorporate them into the current matching 48 ⟩ ≡
  while (dlink) {
    oo, b = tip[dlink], dlink = next[dlink];
    l = final_level;
  enter_level: o, boy[l] = b;
  advance: if (o, blink[b]) {
      ooo, g = tip[blink[b]], blink[b] = next[blink[b]];
      if (o, imate[g] ≡ 0) ⟨ Augment the current matching and continue 49 ⟩;
      if (o, mark[imate[g]] < 0) goto advance;
      b = imate[g], l−−;
      goto enter_level;
    }
    if (++l > final_level) continue;
    o, b = boy[l];
    goto advance;
  }
  ⟨ Reset all marks to zero 47 ⟩;
```

This code is used in section 52.

49. At this point $g = g_0$ and $b = boy[0] = b_0$ in an augmenting path. The other boys are $boy[1]$, $boy[2]$, and so on.

```
⟨ Augment the current matching and continue 49 ⟩ ≡
  {
    if (l) fprintf(stderr, "I'm␣confused!\n");      /* a free girl should occur only at level 0 */
    ⟨ Remove g from the list of free girls 51 ⟩;
    while (1) {
      o, mark[b] = −1;
      ooo, j = mate[b], mate[b] = g, imate[g] = b;
      if (j ≡ 0) break;      /* b was free */
      o, g = j, b = boy[++l];
    }
    continue;
  }
```

This code is used in section 48.

50.
```
⟨ Global variables 4* ⟩ +≡
  int boy[maxn];      /* the boys being explored during the depth-first search */
```

51.
```
⟨ Remove g from the list of free girls 51 ⟩ ≡
  f−−;      /* f is the number of free girls */
  o, j = iqueue[g];      /* where is g in queue? */
  ooo, i = queue[f], queue[j] = i, iqueue[i] = j;      /* OK to clobber queue[f] */
```

This code is used in section 49.

52. Hey folks, we've now got all the infrastructure and machinery of the HK algorithm in place. It only remains to actually perform the algorithm.

```
⟨ Solve that problem and update sol[p][e..e+n−1] 52 ⟩ ≡
  while (1) {
    ⟨ Build the dag of shortest augmenting paths (SAPs) 43 ⟩;
    ⟨ If there are no SAPs, break 46 ⟩;
    ⟨ Find a maximal set of disjoint SAPs, and incorporate them into the current matching 48 ⟩;
  }
  if (f ≡ n − m) ⟨ Store the solution in sol[p] 53 ⟩
  else
  no_sol: for (k = 0; k < n; k++) o, sol[p][e + k] = 0;
  continue;      /* resume the loop on e */
yes_sol: for (k = 0; k < n; k++) o, sol[p][e + k] = 1;
  z += n;
```

This code is used in section 38.

53. The climax. But it's still necessary to don our thinking cap and figure out exactly what we've got, when the HK algorithm has found a perfect matching of m boys to $n > m$ girls.

Our job is to update n entries of sol, one for each girl. That entry should be 0 if and only if the girl has a mate in *every* perfect match. (Because the subgraph isomorphism will assign her to the parent of v in T, while the mated girls will be assigned to some of v's children in the embedding.)

Suppose, for example, that the bipartite matching is unique. In that case we'll want to set $sol[p][g] = 0$ if and only if $imate[g] \neq 0$.

Usually, however, there will be a number of perfect matchings, involving different sets of girls. Matula noticed, in Theorem 3.4 of his paper, that it's actually easy to distinguish the forcibly matched girls from the others. Moreover — fortunately for us — the necessary information is sitting conveniently in the dag, when the HK algorithm ends!

Indeed, it's not difficult to verify that every perfect matching either includes g or corresponds to a path from g to $\perp$ in the dag. Therefore — ta da — the freeable girls are precisely the girls in the first q positions of $queue$!

```
⟨ Store the solution in sol[p] 53 ⟩ ≡
  {
    for (k = 0; k < n; k++) o, sol[p][e + k] = 0;
    for (k = 0; k < q; k++) ooo, z++, sol[p][queue[k]] = 1;
    ⟨ Store the mate information too 54 ⟩;
  }
```

This code is used in section 52.

54. If we're interested only in whether or not an embedding of S into T exists, the sol array tells us everything we need to know.

But if we want to actually see an embedding, we might wish to store the solutions to the matching problems we've solved, so that we don't need to repeat those calculations later.

In a way that's foolish: Only a small number of matching problems will need to be redone. So we're wasting space by storing this extra information — which doesn't fit in sol. And we're gaining only an insignificant amount of time.

Still, the details are interesting, so I'm plunging ahead. Let $solx$ and $soly$ be arrays, such that the solution to the bipartite matching problem in $sol[p][e\,..\,e+n-1]$ is recorded in $solx[p][e\,..\,e+n-1]$ and $soly[p][e\,..\,e+n-1]$. (Both $solx$ and $soly$ are arrays of **int**, while sol itself could have been an array of single bits.)

It suffices to store the final $imate$ table in $solx$, and to store links of a path from g to $\perp$ in $soly$.

```
⟨ Store the mate information too 54 ⟩ ≡
  for (g = e; g < e + n; g++) oo, solx[p][g] = imate[g];
  for (k = 0; k < q; k++) {
    o, g = queue[k];
    if (o, imate[g]) oooo, soly[p][g] = tip[blink[imate[g]]];
  }
```

This code is used in section 53.

55. ⟨ Global variables 4* ⟩ +≡

```
int sol[maxn][maxg];      /* the master control matrix */
int solx[maxn][maxg];     /* imate info for bipartite solutions */
int soly[maxn][maxg];     /* final dag info for bipartite solutions */
```

56. The anticlimax. When all has been done but not yet said, we want to tell the user what happened.

At this point z holds the value of $solve(1)$. It's negative, say $-d$, if the subtree of S rooted at node d and its parent cannot be isomorphically embedded in T. Otherwise z is zero if S itself cannot be embedded, although every subtree of node 1 is embeddable. Otherwise z is the number of arcs e of T for which there's an embedding with node 0 of S mapped into the root of subtree e.

(In the latter case, notice that z is probably *not* the actual total number of embeddings. It's just the number of places where we could start an embedding and obtain at least one success.)

```
⟨ Report the solution 56 ⟩ ≡
  if (z < 0)
    fprintf(stderr, "Failure;␣We␣can't␣even␣embed␣node␣%d␣and␣its␣parent.\n", encode(-z));
  else {
    fprintf(stderr, "There␣%s␣%d␣place%s␣to␣anchor␣an␣embedding␣of␣node␣1.\n",
        z ≡ 1 ? "is" : "are", z, z ≡ 1 ? "" : "s");
    if (z) ⟨ Print a solution 58 ⟩;
  }
```

57. Our final task is to harvest the information in *sol*, *solx*, and *soly*, in order to present the user with the images of nodes 0, 1, ... of S, in one of the possible embeddings found.

To do this, we assign an edge called $solarc[p]$ to each nonroot vertex p of S. If this arc runs from v to u, it means that the embedding maps p to v and p's parent to u. These arcs are assigned top-down, starting with the rightmost e such that $sol[1][e] = 1$.

58.

```
⟨ Print a solution 58 ⟩ ≡
  {
    for (e = emax − 1; o, sol[1][e] ≡ 0; e−−) ;
    oo, solarc[1] = e;
    for (p = 1; p < m; p++)
      if (o, snode[p].child) {
        for (q = snode[p].child; q; o, q = snode[q].sib) o, mate[q] = 0;
        oo, z = solarc[p], v = vert[z];
        o, e = tnode[v].arc, n = tnode[v].deg;
        for (g = e; g < e + n; g++) ooo, q = imate[g] = solx[p][g], mate[q] = g;
        ⟨ Find a matching in which imate[z] = 0 60 ⟩;
        for (o, g = e, q = snode[p].child; q; o, q = snode[q].sib) {
          if (o, mate[q]) oo, solarc[q] = dual[mate[q]];
          else {      /* choose mate for a universally matchable boy */
            while (g ≡ z ∨ (o, imate[g])) g++;
            oo, solarc[q] = dual[g++];
          }
        }
      }
    oo, printf("%c", encode(uert[solarc[1]]));
    for (p = 1; p < m; p++) oo, printf("␣%c", encode(vert[solarc[p]]));
    printf("\n");
  }
```

This code is used in section 56.

59.

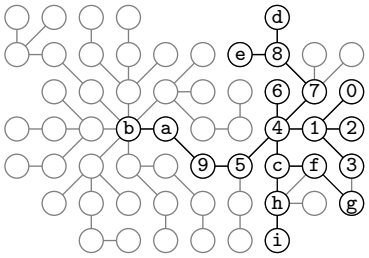
```
⟨ Global variables 4* ⟩ +≡
  int solarc[maxn];      /* key arcs in the solution */
```

60. Here finally is a kind of cute way to end, using the theory of *non*-augmenting paths. (That theory can be understood from the construction of the final, incomplete dag in the HK algorithm, whose critical structure we stored in $soly[p]$.)

```
⟨ Find a matching in which imate[z] = 0 60 ⟩ ≡
  for (k = 0, g = z; o, q = imate[g]; k = q) {
    o, imate[g] = k;
    o, g = soly[p][g];
    o, mate[q] = g;
  }
  o, imate[g] = k;
```

This code is used in section 58.

61. Did you solve the puzzle?



62*. ⟨ Record the solution 62* ⟩ ≡

```
  emems = mems − startmems;
  if (z > 0) oo, msols[mserial]++, nsols[nserial]++, totsols++;
  ⟨ Update the runtime stats 63* ⟩;
```

This code is used in section 2*.

63*. We maintain the mean and variance and max of *emems*, the number of mems elapsed while solving an S-T embedding problem, using Welford's method; see 4.2.2–(16) in *Seminumerical Algorithms*.

```
⟨ Update the runtime stats 63* ⟩ ≡
  {
    register double del;
    samp += 1.0;
    if (emems > ememsmax) ememsmax = emems, shardest = mserial, thardest = nserial;
    del = emems − ememsmean;
    ememsmean += del/samp;
    ememsvar += del ∗ (emems − ememsmean);
  }
```

This code is used in section 62*.

64*. ⟨ Global variables 4* ⟩ +≡

```
  unsigned long long startmems;
  int emems, ememsmax, shardest, thardest;
  double ememsmean, ememsvar, samp;
  int msols[maxtrees], nsols[maxtrees];
  unsigned long long totsols;
  int record;
```

65* #**define** *errorbar*(*x*) ((*x*) ? *sqrt*((*x*)/(*samp* * (*samp* − 1.0))) : 0.0)

⟨ Sign off 65* ⟩ ≡

```
printf("I␣examined␣%d␣%d-trees␣and␣%d␣%d-trees␣(total␣%g␣cases).\n", mserial, m, nserial, n,
    samp);
printf("There␣were␣%lld␣cases␣with␣S␣embeddable␣in␣T.\n", totsols);
printf("Observed␣running␣time␣in␣mems␣was␣%g␣+-␣%g;\n", ememsmean, errorbar(ememsvar));
make_sstring(shardest), make_tstring(thardest);
printf("the␣hardest␣case␣(%d␣mems)␣was␣S=%s␣versus␣T=%s.\n", ememsmax, sstring, tstring);
printf("Here␣are␣extremes␣for␣S␣embeddings:\n");
for (k = p = 0, q = nserial; k < mserial; k++) {
  if (msols[k] ≥ p) p = msols[k], make_sstring(k), printf("␣␣␣%s:%d\n", sstring, p);
  if (msols[k] ≤ q) q = msols[k], make_sstring(k), printf("␣%s:%d\n", sstring, q);
}
printf("Here␣are␣extremes␣for␣T␣embeddings:\n");
for (k = p = 0, q = mserial; k < nserial; k++) {
  if (nsols[k] ≥ p) p = nsols[k], make_tstring(k), printf("␣␣␣%s:%d\n", tstring, p);
  if (nsols[k] ≤ q) q = nsols[k], make_tstring(k), printf("␣%s:%d\n", tstring, q);
}
printf("Altogether␣%lld+%lld␣mems␣for␣this␣computation.\n", imems, mems);
```

This code is used in section 2*.

66* ⟨ Subroutines 5* ⟩ +≡

```
void make_sstring(int k)
{
  register j, t, i, d;
  if (mstart + k < mstop) {
    for (j = mm − 1, t = mstart + k; j; j−−, t = mup[t]) sstring[j] = encode(mp[t]);
  } else {
    d = mshortstop − mshortstart, k −= mstop − mstart;
    for (i = 0; k ≥ d; i++, d−−) k −= d;
    for (j = (mm ≫ 1) − 1, t = mshortstart + i; j; j−−, t = mup[t]) sstring[j] = encode(mp[t]);
    sstring[mm ≫ 1] = '0';
    for (j = mm − 1, t = mshortstart + i + k; j > (mm ≫ 1); j−−, t = mup[t])
      sstring[j] = encode((mm ≫ 1) + mp[t]);
  }
}
void make_tstring(int k)
{
  register j, t, i, d;
  if (nstart + k < nstop) {
    for (j = nn − 1, t = nstart + k; j; j−−, t = up[t]) tstring[j] = encode(np[t]);
  } else {
    d = nshortstop − nshortstart, k −= nstop − nstart;
    for (i = 0; k ≥ d; i++, d−−) k −= d;
    for (j = (nn ≫ 1) − 1, t = nshortstart + i; j; j−−, t = up[t]) tstring[j] = encode(np[t]);
    tstring[nn ≫ 1] = '0';
    for (j = nn − 1, t = nshortstart + i + k; j > (nn ≫ 1); j−−, t = up[t])
      tstring[j] = encode((nn ≫ 1) + np[t]);
  }
}
```

67* Index.

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 39, 62, 63, 64, 65, 66, 67.

MATULA-EXHAUSTIVE