1*. Intro. Here's an easy way to calculate the number of graceful labelings that have m edges and at most n nonisolated vertices, for $0 \le n \le m+1$, given m and n. I subdivide into connected and nonconnected graphs.

The idea is to run through all m-tuples $(x_1, \ldots, x_m)$ with $0 \le x_j \le m - j$; edge j will go from the vertex labeled x_j to the vertex labeled $x_j + j$.

I consider only the labelings in which $x_{m-1} = 1$; in other words, I assume that edge $m - 1$ runs from 1 to m. (These are in one-to-one correspondence with the labelings for which that edge runs from 0 to $m-1$.) But I multiply all the answers by 2; hence the total over all n is exactly $m!$.

I could go through those m-tuples in some sort of Gray code order, with only one x_j changing at a time. But I'm not trying to be tricky or extremely efficient. So I simply use reverse colexicographic order. That is, for each choice of $(x_{j+1}, \ldots, x_m)$, I run through the possibilities for x_j from $m - j$ to 0, in decreasing order.

```
#define maxm 100
```

2. I do, however, want to have some fun with data structures.

Every vertex is represented by its label. Vertex v, for $0 \le v \le m$, is isolated if and only if label v has not been used in any of the edges. (In particular, vertices 0, 1, and m are never isolated, because of the assumption above.)

It's easy to maintain, for each vertex, a linked list of all its neighbors. These lists are stacks, since they change in first-in-last-out fashion.

It's also easy to maintain a dynamic union-find structure, because of the first-in-last-out behavior of this algorithm.

3*. OK, let's get going.

```
#include <stdio.h>
#include <stdlib.h>
  int mm, nn;   /* command-line parameters */
  ⟨Global variables 15⟩;
  main(int argc, char *argv[])
  {
    register j, k, l, m, n;
    ⟨Process the command line 4*⟩;
    ⟨Initialize to (m − 1, ..., 2, 1, 0) 7*⟩;
    while (1) {
      ⟨Study the current graph 16⟩;
      ⟨Move to the next m-tuple, or goto done 5*⟩;
    }
  done: ⟨Print the stats 17*⟩;
  }
```

4*. ⟨ Process the command line 4* ⟩ ≡
```
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);
}
m = mm, n = nn;
if (m < 2 ∨ m > maxm) {
  fprintf(stderr, "Sorry, m must be between 2 and %d!\n", maxm);
  exit(−2);
}
if (n > m + 1) {
  fprintf(stderr, "Sorry, n must be less than m+1\n");
  exit(−3);
}
```
This code is used in section 3*.

5*. ⟨ Move to the next m-tuple, or **goto** *done* 5* ⟩ ≡
```
  for (j = 1; x[j] ≡ 0; j++) {
  tryagain_inloop: ⟨ Delete the edge from x[j] to x[j] + j 9 ⟩;
  }
  if (j ≡ m − 1) goto done;
tryagain: ⟨ Delete the edge from x[j] to x[j] + j 9 ⟩;
  x[j]−−;
  ⟨ Insert an edge from x[j] to x[j] + j 8 ⟩;
  if (active > n) {
    if (x[j] ≡ 0) goto tryagain_inloop;
    else goto tryagain;
  }
  for (j−−; j; j−−) {
    x[j] = m − j;
    ⟨ Insert an edge from x[j] to x[j] + j 8 ⟩;
    if (active > n) goto tryagain;
  }
```
This code is used in section 3*.

6. Graceful structures. An unusual — indeed, somewhat amazing — data structure works well with graceful graphs.

Suppose v has neighbors $w_1, \ldots, w_t$. Let $f_v(w) = w - v$, if $w > v$; $f_v(w) = m + v - w$, if $w < v$. Then we set $arcs[v] = f(w_1)$, or 0 if $t = 0$; $link[f(w_j)] = f(w_{j+1})$ for $1 \le j < t$; and $link[f(w_t)] = 0$.

(Think about it. If $0 < k \le m$, we use $link[k]$ only for an arc from v to $v + k$ for some v. If $m < k \le 2m$, we use $link[k]$ only for an arc from v to $v - (k - m)$ for some v. In either case at most one such arc is present. Thus all of the memory for link storage is preallocated; we don't need a list of available slots.)

7*. We silently use the facts that $arcs[v]$ is initially 0 for all v, and $active = 0$. But the x and $link$ arrays needn't be initialized (I mean, everything would work fine if they were initially garbage).

```
⟨ Initialize to (m − 1, ..., 2, 1, 0) 7* ⟩ ≡
  ⟨ Initialize the union/find structures 11 ⟩;
  for (j = m; j; j−−) {
    x[j] = m − j;
    ⟨ Insert an edge from x[j] to x[j] + j 8 ⟩;
    if (active > n) goto tryagain;
  }
```

This code is used in section 3*.

8.

```
⟨ Insert an edge from x[j] to x[j] + j 8 ⟩ ≡
  {
    register int p, u, v, uu, vv;
    u = x[j];
    v = u + j;
    ⟨ Do a union operation u ≡ v 12 ⟩;
    p = arcs[u];
    if (¬p) active++;
    link[j] = p, arcs[u] = j;
    p = arcs[v];
    if (¬p) active++;
    link[m + j] = p, arcs[v] = m + j;
  }
```

This code is used in sections 5* and 7*.

9.

```
⟨ Delete the edge from x[j] to x[j] + j 9 ⟩ ≡
  {
    register int p, u, v, uu, vv;
    u = x[j];
    v = u + j;
    p = link[m + j];   /* at this point arcs[v] = m + j */
    arcs[v] = p;
    if (¬p) active−−;
    p = link[j];   /* at this point arcs[u] = j */
    arcs[u] = p;
    if (¬p) active−−;
    ⟨ Undo the union operation u ≡ v 14 ⟩;
  }
```

This code is used in section 5*.

10. Two vertices are equivalent if they belong to the same component. We use a classic union-find data structure to keep of equivalences: The invariant relations state that $parent[v] < 0$ and $size[v] = c$ if v is the root of an equivalence class of size c; otherwise $parent[v]$ points to an equivalent vertex that is nearer the root. These trees have at most $\lg m$ levels, because we never merge a tree of size c into a tree of size $< c$.

Variable l is the current number of edges. It is also, therefore, the number of union operations previously done but not yet undone.

11. ⟨ Initialize the union/find structures 11 ⟩ ≡

```
for (j = 0; j ≤ m; j++) parent[j] = −1, size[j] = 1;    /* and l = 0 */
l = 0;
```

This code is used in section 7*.

12. ⟨ Do a union operation $u \equiv v$ 12 ⟩ ≡

```
for (uu = u; parent[uu] ≥ 0; uu = parent[uu]) ;
for (vv = v; parent[vv] ≥ 0; vv = parent[vv]) ;
if (uu ≡ vv) move[l] = −1;
else if (size[uu] ≤ size[vv]) parent[uu] = vv, move[l] = uu, size[vv] += size[uu];
else parent[vv] = uu, move[l] = vv, size[uu] += size[vv];
l++;
```

This code is used in section 8.

13. Dynamic union-find is ridiculously easy because, as observed above, the operations are strictly last-in-first-out. And we didn't clobber the *size* information when merging two classes.

14. ⟨ Undo the union operation $u \equiv v$ 14 ⟩ ≡

```
l−−;
uu = move[l];
if (uu ≥ 0) {
  vv = parent[uu];    /* we have parent[vv] < 0 */
  size[vv] −= size[uu];
  parent[uu] = −1;
}
```

This code is used in section 9.

15. ⟨ Global variables 15 ⟩ ≡

```
int active;    /* this many vertices are currently labeled (not isolated) */
int parent[maxm + 1], size[maxm + 1], move[maxm];    /* the union-find structures */
int arcs[maxm + 1];    /* the first neighbor of v */
int link[2 ∗ maxm + 1];    /* the next element in a list of neighbors */
int x[maxm + 1];    /* the governing sequence of edge choices */
```

See also section 18.

This code is used in section 3*.

16. Doing it. Now we're ready to harvest the routines we've built up.

[A puzzle for the reader: Is *parent*[m] always negative at this point? Answer: Not if, say, $m = 7$ and $(x_1, \ldots, x_m) = (5, 4, 3, 2, 0, 1, 0)$.]

⟨ Study the current graph 16 ⟩ ≡

```
for (k = parent[m]; parent[k] ≥ 0; k = parent[k]) ;
if (size[k] ≡ active) connected[active]++;
else disconnected[active]++;
```

This code is used in section 3*.

17*. ⟨ Print the stats 17* ⟩ ≡

```
printf("Counts␣for␣%d␣edges␣and␣at␣most␣%d␣vertices:\n", m, n);
for (k = 2; k ≤ m + 1; k++)
  if (connected[k] + disconnected[k]) {
    printf("on␣%5d␣vertices,␣%lld␣are␣connected,␣%lld␣not\n", k, 2 ∗ connected[k],
        2 ∗ disconnected[k]);
    totconnected += 2 ∗ connected[k], totdisconnected += 2 ∗ disconnected[k];
  }
printf("Altogether␣%lld␣connected␣and␣%lld␣not.\n", totconnected, totdisconnected);
```

This code is used in section 3*.

18. ⟨ Global variables 15 ⟩ +≡

```
unsigned long long connected[maxm + 2], disconnected[maxm + 2];
unsigned long long totconnected, totdisconnected;
```

19*. Index.

The following sections were changed by the change file: 1, 3, 4, 5, 7, 17, 19.

GRACEFUL-COUNT-SMALL