1. Intro. This program finds all cycles of length k in a given graph, using brute force.

More precisely, the task is to find a sequence of distinct vertices $(v_0, v_1, \ldots, v_{k-1})$ such that $v_{i-1} — v_i$ for $1 \le i < k$ and $v_{k-1} — v_0$. To avoid duplicates, I also require that $v_0 = \max v_i$ and that v_{k-1} precedes v_1 on the adjacency list of v_0. Straightforwarding backtracking is used to run through all of these possibilities.

```
#define maxn 100  /* upper bound on vertices in the graph */
#include <stdio.h>
#include <stdlib.h>
#include "gb_graph.h"
#include "gb_save.h"
  int kk;  /* the given cycle length */
  Vertex *vv[maxn];  /* tentative cycle */
  Arc *aa[maxn];  /* pointers to them the adjacency lists */
  long count;  /* the number of cycles found */

  main(int argc, char *argv[])
  {
    register int i, j, k;
    register Graph *g;
    register Vertex *u, *v;
    register Arc *a, *b;
    Vertex *v0;

    ⟨Process the command line 2⟩;
    ⟨Clear the eligibility tags 5⟩;
    for (v0 = g→vertices + g→n − 1; v0 ≥ g→vertices; v0 −−) ⟨Print all cycles whose largest vertex is v0 3⟩;
    fprintf(stderr, "Altogether␣%ld␣cycles␣found.\n", count);
  }
```

2. ⟨Process the command line 2⟩ ≡

```
  if (argc ≠ 3 ∨ sscanf(argv[2], "%d", &kk) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣foo.gb␣k\n", argv[0]);
    exit(−1);
  }
  g = restore_graph(argv[1]);
  if (¬g) {
    fprintf(stderr, "I␣couldn't␣reconstruct␣graph␣%s!\n", argv[1]);
    exit(−2);
  }
  if (g→n > maxn) {
    fprintf(stderr, "Recompile␣me:␣g->n=%ld,␣maxn=%d!\n", g→n, maxn);
    exit(−3);
  }
  if (kk < 3) {
    fprintf(stderr, "The␣cycle␣length␣must␣be␣3␣or␣more,␣not␣%d!\n", kk);
    exit(−4);
  }
```

This code is used in section 1.

3. **#define** *elig* *u.I* /* is this vertex a legal candidate for v_{k-1}? */

⟨ Print all cycles whose largest vertex is *v0* 3 ⟩ ≡

```
{
  vv[0] = v0;
  for (v = g→vertices; v < v0; v++) v→elig = 0;
  for (a = v→arcs; a; a = a→next)
    if (a→tip < v0) break;
  if (a ≡ 0) continue;    /* reject v0 if it has no smaller neighbors */
  aa[1] = a, k = 1;
try_again: if (k ≡ 1) aa[1]→tip→elig = 1;
  for (a = aa[k]→next; a; a = a→next)
    if (a→tip < v0) break;
tryit: if (a ≡ 0) goto backtrack;
  aa[k] = a, vv[k] = v = a→tip;
  for (j = 0; vv[j] ≠ v; j++) ;
  if (j < k) goto try_again;    /* v is already present */
  k++;
new_level: if (k ≡ kk) ⟨ Check for a solution, then backtrack 4 ⟩;
  for (a = vv[k − 1]→arcs; a; a = a→next)
    if (a→tip < v0) break;
  goto tryit;
backtrack: if (−−k) goto try_again;
}
```

This code is used in section 1.

4. At this point I use the slightly tricky fact that $v = vv[k-1]$.

⟨ Check for a solution, then backtrack 4 ⟩ ≡

```
{
  if (v→elig) {
    for (j = 0; j < kk; j++) printf("␣%s", vv[j]→name);
    printf("\n");
    count++;
  }
  goto backtrack;
}
```

This code is used in section 3.

5. I've avoided tricks, except in one respect that could have caused a bug: The code above assumes that *v*→*elig* is zero for all $v \geq v0$.

That assumption will be valid if we make sure that it holds the first time, since *v0* continues to decrease.

⟨ Clear the eligibility tags 5 ⟩ ≡

```
(g→vertices + g→n − 1)→elig = 0;
```

This code is used in section 1.

6. Index.

GRAPH-CYC