Important: Before reading GB_RAND, please read or at least skim the program for GB_GRAPH.

1. Random graphs. This GraphBase module provides two external subroutines called *random_graph* and *random_bigraph*, which generate graphs in which the arcs or edges have been selected "at random." A third subroutine, *random_lengths*, randomizes the lengths of the arcs of a given graph. The performance of algorithms on such graphs can fruitfully be compared to their performance on the nonrandom graphs generated by other GraphBase routines.

Before reading this code, the reader should be familiar with the basic data structures and conventions described in GB_GRAPH. The routines in GB_GRAPH are loaded together with all GraphBase applications, and the programs below are typical illustrations of how to use them.

```
#define random_graph r_graph /* abbreviations for Procrustean external linkage */
#define random_bigraph r_bigraph
#define random_lengths r_lengths
```

⟨ gb_rand.h 1 ⟩ ≡

```
#define random_graph r_graph /* users of GB_RAND should include this header info */
#define random_bigraph r_bigraph
#define random_lengths r_lengths
  extern Graph *random_graph();
  extern Graph *random_bigraph();
  extern long random_lengths();
```

2. Here is an overview of the file `gb_rand.c`, the C code for the routines in question.

```
#include "gb_graph.h" /* this header file teaches C about GraphBase */
#include "gb_flip.h" /* we will use the GB_FLIP routines for random numbers */
  ⟨ Preprocessor definitions ⟩
  ⟨ Private declarations 8 ⟩
  ⟨ Internal functions 18 ⟩
  ⟨ External functions 5 ⟩
```

3. The procedure *random_graph*(n, m, *multi*, *self*, *directed*, *dist_from*, *dist_to*, *min_len*, *max_len*, *seed*) is designed to produce a pseudo-random graph with n vertices and m arcs or edges, using pseudo-random numbers that depend on *seed* in a system-independent fashion. The remaining parameters specify a variety of options:

multi $\neq 0$ permits duplicate arcs;
self $\neq 0$ permits self-loops (arcs from a vertex to itself);
directed $\neq 0$ makes the graph directed; otherwise each arc becomes an undirected edge;
dist_from and *dist_to* specify probability distributions on the arcs;
min_len and *max_len* bound the arc lengths, which will be uniformly distributed between these limits.

If *dist_from* or *dist_to* are Λ, the probability distribution is uniform over vertices; otherwise the *dist* parameter points to an array of n nonnegative integers that sum to 2^{30}, specifying the respective probabilities (times 2^{30}) that each given vertex will appear as the source or destination of the random arcs.

A special option *multi* $= -1$ is provided. This acts exactly like *multi* $= 1$, except that arcs are not physically duplicated in computer memory—they are replaced by a single arc whose length is the minimum of all arcs having a common source and destination.

The vertices are named simply `"0"`, `"1"`, `"2"`, and so on.

4. Examples: $random\_graph(1000, 5000, 0, 0, 0, \Lambda, \Lambda, 1, 1, 0)$ creates a random undirected graph with 1000 vertices and 5000 edges (hence 10000 arcs) of length 1, having no duplicate edges or self-loops. There are $\binom{1000}{2} = 499500$ possible undirected edges on 1000 vertices; hence there are exactly $\binom{499500}{5000}$ possible graphs meeting these specifications. Every such graph would be equally likely, if *random_graph* had access to an ideal source of random numbers. The GraphBase programs are designed to be system-independent, so that identical graphs will be obtained by everybody who asks for $random\_graph(1000, 5000, 0, 0, 0, \Lambda, \Lambda, 1, 1, 0)$. Equivalent experiments on algorithms for graph manipulation can therefore be performed by researchers in different parts of the world.

The subroutine call $random\_graph(1000, 5000, 0, 0, 0, \Lambda, \Lambda, 1, 1, s)$ will produce different graphs when the random seed s varies; however, the graph for any particular value of s will be the same on all computers. The seed value can be any integer in the range $0 \le s < 2^{31}$.

To get a random directed graph, allowing self-loops and repeated arcs, and with a uniform distribution on vertices, ask for

$$random\_graph(n, m, 1, 1, 1, \Lambda, \Lambda, 1, 1, s).$$

Each of the m arcs of that digraph has probability $1/n^2$ of being from u to v, for all u and v. If self-loops are disallowed (by changing '$1, 1, 1$' to '$1, 0, 1$'), each arc has probability $1/(n^2 - n)$ of being from u to v, for all $u \ne v$.

To get a random directed graph in which vertex k is twice as likely as vertex $k + 1$ to be the source of an arc but only half as likely to be the destination of an arc, for all k, try

$$random\_graph(31, m, 1, 1, 1, d0, d1, 0, 255, s)$$

where the arrays $d0$ and $d1$ have the static declarations

$$\begin{aligned}&\textbf{long}\ d0[31] = \{\#\mathtt{20000000}, \#\mathtt{10000000}, \ldots, 4, 2, 1, 1\};\\ &\textbf{long}\ d1[31] = \{1, 1, 2, 4, \ldots, \#\mathtt{10000000}, \#\mathtt{20000000}\};\end{aligned}$$

then about $1/4$ of the arcs will run from 0 to 30, while arcs from 30 to 0 will be extremely rare (occurring with probability 2^{-60}). Incidentally, the arc lengths in this example will be random bytes, uniformly distributed between 0 and 255, because $min\_len = 0$ and $max\_len = 255$.

If we forbid repeated arcs in this example, by setting $multi = 0$, the effect is to discard all arcs having the same source and destination as a previous arc, regardless of length. In such a case m had better not be too large, because the algorithm will keep going until it has found m distinct arcs, and many arcs are quite rare indeed; they will probably not be found until hundreds of centuries have elapsed.

A random bipartite graph can also be obtained as a special case of *random_graph*; this case is explained below.

Semantics: If $multi = directed = 0$ and $self \ne 0$, we have an undirected graph without duplicate edges but with self-loops permitted. A self-loop then consists of two identical self-arcs, in spite of the fact that $multi = 0$.

5. If the *random_graph* routine encounters a problem, it returns Λ, after putting a code number into the external variable *panic_code*. This code number identifies the type of failure. Otherwise *random_graph* returns a pointer to the newly created graph and leaves *panic_code* unchanged. The *gb_trouble_code* will be cleared to zero after *random_graph* has acted.

```
#define panic(c) { panic_code = c; gb_trouble_code = 0; return Λ; }
⟨External functions 5⟩ ≡
  Graph *random_graph(n, m, multi, self, directed, dist_from, dist_to, min_len, max_len, seed)
      unsigned long n;    /* number of vertices desired */
      unsigned long m;    /* number of arcs or edges desired */
      long multi;    /* allow duplicate arcs? */
      long self;    /* allow self loops? */
      long directed;    /* directed graph? */
      long *dist_from;    /* distribution of arc sources */
      long *dist_to;    /* distribution of arc destinations */
      long min_len, max_len;    /* bounds on random lengths */
      long seed;    /* random number seed */
  { ⟨Local variables 6⟩
    if (n ≡ 0) panic(bad_specs);    /* we gotta have a vertex */
    if (min_len > max_len) panic(very_bad_specs);    /* what are you trying to do? */
    if (((unsigned long)(max_len)) − ((unsigned long)(min_len)) ≥ ((unsigned long) #80000000))
      panic(bad_specs + 1);    /* too much range */
    ⟨Check the distribution parameters 11⟩;
    gb_init_rand(seed);
    ⟨Create a graph with n vertices and no arcs 7⟩;
    ⟨Build tables for nonuniform distributions, if needed 13⟩;
    for (mm = m; mm; mm−−) ⟨Add a random arc or a random edge 9⟩;
  trouble:
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);    /* oops, we ran out of memory somewhere back there */
    }
    gb_free(new_graph→aux_data);
    return new_graph;
  }
```

See also sections 22 and 24.

This code is used in section 2.

6. ⟨Local variables 6⟩ ≡

```
  Graph *new_graph;    /* the graph constructed by random_graph */
  long mm;    /* the number of arcs or edges we still need to generate */
  register long k;    /* vertex being processed */
```

See also section 12.

This code is used in section 5.

7. **#define** *dist_code*(x) (x ? "dist" : "0")

⟨ Create a graph with n vertices and no arcs 7 ⟩ ≡

```
new_graph = gb_new_graph(n);
if (new_graph ≡ Λ) panic(no_room);      /* out of memory before we're even started */
for (k = 0; k < n; k++) {
  sprintf(name_buffer, "%ld", k);
  (new_graph→vertices + k)→name = gb_save_string(name_buffer);
}
sprintf(new_graph→id, "random_graph(%lu,%lu,%d,%d,%d,%s,%s,%ld,%ld,%ld)",
    n, m, multi > 0 ? 1 : multi < 0 ? −1 : 0, self ? 1 : 0, directed ? 1 : 0,
    dist_code(dist_from), dist_code(dist_to), min_len, max_len, seed);
```

This code is used in section 5.

8. ⟨ Private declarations 8 ⟩ ≡

```
static char name_buffer[] = "9999999999";
```

See also sections 14, 17, and 25.

This code is used in section 2.

9. **#define** *rand_len* (*min_len* ≡ *max_len* ? *min_len* : *min_len* + *gb_unif_rand*(*max_len* − *min_len* + 1))

⟨ Add a random arc or a random edge 9 ⟩ ≡

```
{ register Vertex *u, *v;
repeat:
  if (dist_from) ⟨ Generate a random vertex u according to dist_from 15 ⟩
  else u = new_graph→vertices + gb_unif_rand(n);
  if (dist_to) ⟨ Generate a random vertex v according to dist_to 16 ⟩
  else v = new_graph→vertices + gb_unif_rand(n);
  if (u ≡ v ∧ ¬self) goto repeat;
  if (multi ≤ 0) ⟨ Search for duplicate arcs or edges; goto repeat or done if found 10 ⟩;
  if (directed) gb_new_arc(u, v, rand_len);
  else gb_new_edge(u, v, rand_len);
done: ;
}
```

This code is used in section 5.

10. When we decrease the length of an existing edge, we use the fact that its two arcs are adjacent in memory. If $u \equiv v$ in this case, we encounter the first of two mated arcs before seeing the second; hence the mate of the arc we find is in location $a+1$ when $u \le v$, and in location $a-1$ when $u > v$.

We must exit to location *trouble* if memory has been exhausted; otherwise there is a danger of an infinite loop, with *dummy_arc→next* = *dummy_arc*.

⟨ Search for duplicate arcs or edges; **goto** *repeat* or *done* if found 10 ⟩ ≡

```
if (gb_trouble_code) goto trouble;
else { register Arc *a;
  long len;    /* length of new arc or edge being combined with previous */
  for (a = u→arcs; a; a = a→next)
    if (a→tip ≡ v)
      if (multi ≡ 0) goto repeat;    /* reject a duplicate arc */
      else {    /* multi < 0 */
        len = rand_len;
        if (len < a→len) {
          a→len = len;
          if (¬directed) {
            if (u ≤ v) (a + 1)→len = len;
            else (a − 1)→len = len;
          }
        }
        goto done;
      }
}
```

This code is used in section 9.

11. Nonuniform random number generation. The *random_graph* procedure is complete except for the parts that handle general distributions *dist_from* and *dist_to*. Before attempting to generate those distributions, we had better check them to make sure that the specifications are well formed; otherwise disaster might ensue later. This part of the program is easy.

⟨ Check the distribution parameters 11 ⟩ ≡

```
{ register long acc;   /* sum of probabilities */
  register long *p;   /* pointer to current probability of interest */
  if (dist_from) {
    for (acc = 0, p = dist_from; p < dist_from + n; p++) {
      if (*p < 0) panic(invalid_operand);   /* dist_from contains a negative entry */
      if (*p > #40000000 − acc) panic(invalid_operand + 1);   /* probability too high */
      acc += *p;
    }
    if (acc ≠ #40000000) panic(invalid_operand + 2);   /* dist_from table doesn't sum to 2^30 */
  }
  if (dist_to) {
    for (acc = 0, p = dist_to; p < dist_to + n; p++) {
      if (*p < 0) panic(invalid_operand + 5);   /* dist_to contains a negative entry */
      if (*p > #40000000 − acc) panic(invalid_operand + 6);   /* probability too high */
      acc += *p;
    }
    if (acc ≠ #40000000) panic(invalid_operand + 7);   /* dist_to table doesn't sum to 2^30 */
  }
}
```

This code is used in section 5.

12. We generate nonuniform distributions by using Walker's alias method (see, for example, *Seminumerical Algorithms*, second edition, exercise 3.4.1–7). Walker's method involves setting up "magic" tables of length nn, where nn is the smallest power of 2 that is $\geq n$.

format *magic_entry int*

⟨ Local variables 6 ⟩ +≡

```
long nn = 1;   /* this will be increased to 2^⌈lg n⌉ */
long kk = 31;   /* this will be decreased to 31 − ⌈lg n⌉ */
magic_entry *from_table, *to_table;   /* alias tables */
```

13. ⟨ Build tables for nonuniform distributions, if needed 13 ⟩ ≡

```
{
  if (dist_from) {
    while (nn < n) nn += nn, kk−−;
    from_table = walker(n, nn, dist_from, new_graph);
  }
  if (dist_to) {
    while (nn < n) nn += nn, kk−−;
    to_table = walker(n, nn, dist_to, new_graph);
  }
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);   /* oops, we ran out of memory somewhere back there */
  }
}
```

This code is used in section 5.

14. ⟨ Private declarations 8 ⟩ +≡

```
typedef struct {
  long prob;    /* a probability, multiplied by 2^31 and translated */
  long inx;    /* index that might be selected */
} magic_entry;
```

15. Once the magic tables have been set up, we can generate nonuniform vertices by using the following code:

⟨ Generate a random vertex *u* according to *dist_from* 15 ⟩ ≡

```
{ register magic_entry *magic;
  register long uu = gb_next_rand();    /* uniform random number */
  k = uu >> kk;
  magic = from_table + k;
  if (uu <= magic->prob) u = new_graph->vertices + k;
  else u = new_graph->vertices + magic->inx;
}
```

This code is used in section 9.

16. ⟨ Generate a random vertex *v* according to *dist_to* 16 ⟩ ≡

```
{ register magic_entry *magic;
  register long uu = gb_next_rand();    /* uniform random number */
  k = uu >> kk;
  magic = to_table + k;
  if (uu <= magic->prob) v = new_graph->vertices + k;
  else v = new_graph->vertices + magic->inx;
}
```

This code is used in section 9.

17. So all we have to do is set up those magic tables. If *uu* is a uniform random integer between 0 and $2^{31}-1$, the index $k = uu \gg kk$ is a uniform random integer between 0 and $nn-1$, because of the relation between *nn* and *kk*. Once k is computed, the code above selects vertex k with probability $(p+1-(k \ll kk))/2^{31}$, where $p = magic\to prob$ and *magic* is the kth element of the magic table; otherwise the code selects vertex $magic\to inx$. The trick is to set things up so that each vertex is selected with the proper overall probability.

Let's imagine that the given distribution vector has length *nn*, instead of n, by extending it if necessary with zeroes. Then the average entry among these *nn* integers is exactly $t = 2^{30}/nn$. If some entry, say entry i, exceeds t, there must be another entry that's less than t, say entry j. We can set the jth entry of the magic table so that its *prob* field selects vertex j with the correct probability, and so that its *inx* field equals i. Then we are selecting vertex i with a certain residual probability; so we subtract that residual from i's present probability, and repeat the process with vertex j eliminated. The average of the remaining entries is still t, so we can repeat this procedure until all remaining entries are exactly equal to t. The rest is easy.

During the calculation, we maintain two linked lists of (*prob*, *inx*) pairs. The *hi* list contains entries with $prob > t$, and the *lo* list contains the rest. During this part of the computation we call these list elements 'nodes', and we use the field names *key* and *j* instead of *prob* and *inx*.

```
⟨ Private declarations 8 ⟩ +≡
  typedef struct node_struct {
    long key;    /* a numeric quantity */
    struct node_struct *link;    /* the next node on the list */
    long j;    /* a vertex number to be selected with probability key/2^30 */
  } node;
  static Area temp_nodes;    /* nodes will be allocated in this area */
  static node *base_node;    /* beginning of a block of nodes */
```

18.

```
⟨ Internal functions 18 ⟩ ≡
  static magic_entry *walker(n, nn, dist, g)
      long n;    /* length of dist vector */
      long nn;    /* 2^⌈lg n⌉ */
      register long *dist;    /* start of distribution table, which sums to 2^30 */
      Graph *g;    /* tables will be allocated for this graph's vertices */
  { magic_entry *table;    /* this will be the magic table we compute */
    long t;    /* average key value */
    node *hi = Λ, *lo = Λ;    /* nodes not yet included in magic table */
    register node *p, *q;    /* pointer variables for list manipulation */
    base_node = gb_typed_alloc(nn, node, temp_nodes);
    table = gb_typed_alloc(nn, magic_entry, g→aux_data);
    if (¬gb_trouble_code) {
      ⟨ Initialize the hi and lo lists 19 ⟩;
      while (hi) ⟨ Remove a lo element and match it with a hi element; deduct the residual probability
            from that hi element 20 ⟩;
      while (lo) ⟨ Remove a lo element of key value t 21 ⟩;
    }
    gb_free(temp_nodes);
    return table;    /* if gb_trouble_code is nonzero, the table is empty */
  }
```

This code is used in section 2.

19. ⟨ Initialize the *hi* and *lo* lists 19 ⟩ ≡

```
t = #40000000/nn;   /* this division is exact */
p = base_node;
while (nn > n) {
  p→key = 0;
  p→link = lo;
  p→j = −−nn;
  lo = p++;
}
for (dist = dist + n − 1; n > 0; dist−−, p++) {
  p→key = ∗dist;
  p→j = −−n;
  if (∗dist > t) p→link = hi, hi = p;
  else p→link = lo, lo = p;
}
```

This code is used in section 18.

20. When we change the scale factor from 2^{30} to 2^{31}, we need to be careful lest integer overflow occur. The introduction of register x into this code removes the risk.

⟨ Remove a *lo* element and match it with a *hi* element; deduct the residual probability from that *hi* element 20 ⟩ ≡

```
{ register magic_entry ∗r;
  register long x;
  p = hi, hi = p→link;
  q = lo, lo = q→link;
  r = table + q→j;
  x = t ∗ q→j + q→key − 1;
  r→prob = x + x + 1;
  r→inx = p→j;
    /* we have just given q→key units of probability to vertex q→j, and t − q→key units to vertex p→j */
  if ((p→key −= t − q→key) > t) p→link = hi, hi = p;
  else p→link = lo, lo = p;
}
```

This code is used in section 18.

21. When all remaining entries have the average probability, the *inx* component need not be set, because it will never be used.

⟨ Remove a *lo* element of *key* value t 21 ⟩ ≡

```
{ register magic_entry ∗r;
  register long x;
  q = lo, lo = q→link;
  r = table + q→j;
  x = t ∗ q→j + t − 1;
  r→prob = x + x + 1;   /* that's t units of probability for vertex q→j */
}
```

This code is used in section 18.

22. Random bipartite graphs. The procedure call

$$random\_bigraph(n1, n2, m, multi, dist1, dist2, min\_len, max\_len, seed)$$

is designed to produce a pseudo-random bipartite graph with *n1* vertices in one part and *n2* in the other, having *m* edges. The remaining parameters *multi*, *dist1*, *dist2*, *min_len*, *max_len*, and *seed* have the same meaning as the analogous parameters of *random_graph*.

In fact, *random_bigraph* does its work by reducing its parameters to a special case of *random_graph*. Almost all that needs to be done is to pad *dist1* with *n2* trailing zeroes and *dist2* with *n1* leading zeroes. The only slightly tricky part occurs when *dist1* and/or *dist2* are null, since non-null distribution vectors summing exactly to 2^{30} must then be fabricated.

```
⟨External functions 5⟩ +≡
  Graph *random_bigraph(n1, n2, m, multi, dist1, dist2, min_len, max_len, seed)
      unsigned long n1, n2;   /* number of vertices desired in each part */
      unsigned long m;   /* number of edges desired */
      long multi;   /* allow duplicate edges? */
      long *dist1, *dist2;   /* distribution of edge endpoints */
      long min_len, max_len;   /* bounds on random lengths */
      long seed;   /* random number seed */
  { unsigned long n = n1 + n2;   /* total number of vertices */
    Area new_dists;
    long *dist_from, *dist_to;
    Graph *new_graph;

    init_area(new_dists);
    if (n1 ≡ 0 ∨ n2 ≡ 0) panic(bad_specs);   /* illegal options */
    if (min_len > max_len) panic(very_bad_specs);   /* what are you trying to do? */
    if (((unsigned long)(max_len)) − ((unsigned long)(min_len)) ≥ ((unsigned long) #80000000))
      panic(bad_specs + 1);   /* too much range */
    dist_from = gb_typed_alloc(n, long, new_dists);
    dist_to = gb_typed_alloc(n, long, new_dists);
    if (gb_trouble_code) {
      gb_free(new_dists);
      panic(no_room + 2);   /* no room for auxiliary distribution tables */
    }
    ⟨Compute the entries of dist_from and dist_to 23⟩;
    new_graph = random_graph(n, m, multi, 0L, 0L, dist_from, dist_to, min_len, max_len, seed);
    sprintf(new_graph→id, "random_bigraph(%lu,%lu,%lu,%d,%s,%s,%ld,%ld,%ld)",
        n1, n2, m, multi > 0 ? 1 : multi < 0 ? −1 : 0, dist_code(dist1), dist_code(dist2),
        min_len, max_len, seed);
    mark_bipartite(new_graph, n1);
    gb_free(new_dists);
    return new_graph;
  }
```

23. The relevant identity we need here is the replicative law for the floor function:

$$\left\lfloor \frac{x}{n} \right\rfloor + \left\lfloor \frac{x+1}{n} \right\rfloor + \cdots + \left\lfloor \frac{x+n-1}{n} \right\rfloor = \lfloor x \rfloor .$$

⟨ Compute the entries of *dist_from* and *dist_to* 23 ⟩ ≡

```
{ register long *p, *q;   /* traversers of the dists */
  register long k;   /* vertex count */
  p = dist1;
  q = dist_from;
  if (p)
    while (p < dist1 + n1) *q++ = *p++;
  else
    for (k = 0; k < n1; k++) *q++ = (#40000000 + k)/n1;
  p = dist2;
  q = dist_to + n1;
  if (p)
    while (p < dist2 + n2) *q++ = *p++;
  else
    for (k = 0; k < n2; k++) *q++ = (#40000000 + k)/n2;
}
```

This code is used in section 22.

24. Random lengths. The subroutine call

$$random\_lengths(g, directed, min\_len, max\_len, dist, seed)$$

takes an existing graph and assigns new lengths to each of its arcs. If $dist = \Lambda$, the lengths will be uniformly distributed between *min_len* and *max_len* inclusive; otherwise *dist* should be a probability distribution vector of length $max\_len - min\_len + 1$, like those in *random_graph*.

If $directed = 0$, pairs of arcs $u \to v$ and $v \to u$ will be regarded as a single edge, both arcs receiving the same length.

The procedure returns a nonzero value if something goes wrong; in that case, graph g will not have been changed.

Alias tables for generating nonuniform random lengths will survive in *g→aux_data*.

```
⟨External functions 5⟩ +≡
  long random_lengths(g, directed, min_len, max_len, dist, seed)
      Graph *g;    /* graph whose lengths will be randomized */
      long directed;    /* is it directed? */
      long min_len, max_len;    /* bounds on random lengths */
      long *dist;    /* distribution of lengths */
      long seed;    /* random number seed */
  { register Vertex *u, *v;    /* current vertices of interest */
    register Arc *a;    /* current arc of interest */
    long nn = 1, kk = 31;    /* variables for nonuniform generation */
    magic_entry *dist_table;    /* alias table for nonuniform generation */
    if (g ≡ Λ) return missing_operand;    /* where is g? */
    gb_init_rand(seed);
    if (min_len > max_len) return very_bad_specs;    /* what are you trying to do? */
    if (((unsigned long)(max_len)) − ((unsigned long)(min_len)) ≥ ((unsigned long) #80000000))
      return bad_specs;    /* too much range */
    ⟨Check dist for validity, and set up the dist_table 26⟩;
    sprintf(buffer, ",%d,%ld,%ld,%s,%ld)", directed ? 1 : 0,
        min_len, max_len, dist_code(dist), seed);
    make_compound_id(g, "random_lengths(", g, buffer);
    ⟨Run through all arcs and assign new lengths 27⟩;
    return 0;
  }
```

25. ⟨Private declarations 8⟩ +≡

```
  static char buffer[] = "1,-1000000001,-1000000000,dist,1000000000)";
```

26. ⟨ Check *dist* for validity, and set up the *dist_table* 26 ⟩ ≡

```
if (dist) { register long acc;    /* sum of probabilities */
  register long *p;    /* pointer to current probability of interest */
  register long n = max_len - min_len + 1;

  for (acc = 0, p = dist; p < dist + n; p++) {
    if (*p < 0) return -1;    /* negative probability */
    if (*p > #40000000 - acc) return 1;    /* probability too high */
    acc += *p;
  }
  if (acc ≠ #40000000) return 2;    /* probabilities don't sum to 1 */
  while (nn < n) nn += nn, kk--;
  dist_table = walker(n, nn, dist, g);
  if (gb_trouble_code) {
    gb_trouble_code = 0;
    return alloc_fault;    /* not enough room to generate the magic tables */
  }
}
```

This code is used in section 24.

27. ⟨ Run through all arcs and assign new lengths 27 ⟩ ≡

```
for (u = g→vertices; u < g→vertices + g→n; u++)
  for (a = u→arcs; a; a = a→next) {
    v = a→tip;
    if (directed ≡ 0 ∧ u > v) a→len = (a - 1)→len;
    else { register long len;    /* a random length */
      if (dist ≡ 0) len = rand_len;
      else { long uu = gb_next_rand();
        long k = uu ≫ kk;
        magic_entry *magic = dist_table + k;

        if (uu ≤ magic→prob) len = min_len + k;
        else len = min_len + magic→inx;
      }
      a→len = len;
      if (directed ≡ 0 ∧ u ≡ v ∧ a→next ≡ a + 1) (++a)→len = len;
    }
  }
```

This code is used in section 24.

28. Index. Here is a list that shows where the identifiers of this program are defined and used.

August 4, 2025 at 09:09

GB_RAND

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.