Important: Before reading GB_LISA, please read or at least skim the programs for GB_GRAPH and GB_IO.

1. Introduction. This GraphBase module contains the *lisa* subroutine, which creates rectangular matrices of data based on Leonardo da Vinci's *Gioconda* (aka Mona Lisa). It also contains the *plane_lisa* subroutine, which constructs undirected planar graphs based on *lisa*, and the *bi_lisa* subroutine, which constructs undirected bipartite graphs. Another example of the use of *lisa* can be found in the demo program ASSIGN_LISA.

```
#define plane_lisa p_lisa   /* abbreviation for Procrustean external linkage */
```

⟨ gb_lisa.h 1 ⟩ ≡

```
#define plane_lisa p_lisa
  extern long *lisa();
  extern Graph *plane_lisa();
  extern Graph *bi_lisa();
```

See also sections 3 and 25.

2. The subroutine call $lisa(m, n, d, m0, m1, n0, n1, d0, d1, area)$ constructs an $m \times n$ matrix of integers in the range $[0\,..\,d]$, based on the information in `lisa.dat`. Storage space for the matrix is allocated in the memory area called *area*, using the normal GraphBase conventions explained in GB_GRAPH. The entries of the matrix can be regarded as pixel data, with 0 representing black and d representing white, and with intermediate values representing shades of gray.

The data in `lisa.dat` has 360 rows and 250 columns. The rows are numbered 0 to 359 from top to bottom, and the columns are numbered 0 to 249 from left to right. The output of *lisa* is generated from a rectangular section of the picture consisting of $m1 - m0$ rows and $n1 - n0$ columns; more precisely, *lisa* uses the data in positions (k, l) for $m0 \le k < m1$ and $n0 \le l < n1$.

One way to understand the process of mapping $M = m1 - m0$ rows and $N = n1 - n0$ columns of input into m rows and n columns of output is to imagine a giant matrix of mM rows and nN columns in which the original input data has been replicated as an $M \times N$ array of submatrices of size $m \times n$; each of the submatrices contains mn identical pixel values. We can also regard the giant matrix as an $m \times n$ array of submatrices of size $M \times N$. The pixel values to be output are obtained by averaging the MN pixel values in the submatrices of this second interpretation.

More precisely, the output pixel value in a given row and column is obtained in two steps. First we sum the MN entries in the corresponding submatrix of the giant matrix, obtaining a value D between 0 and $255MN$. Then we scale the value D linearly into the desired final range $[0\,..\,d]$ by setting the result to 0 if $D < d0$, to d if $D \ge d1$, and to $\lfloor d(D - d0)/(d1 - d0) \rfloor$ if $d0 \le D < d1$.

```
#define MAX_M 360   /* the total number of rows of input data */
#define MAX_N 250   /* the total number of columns of input data */
#define MAX_D 255   /* maximum pixel value in the input data */
```

3. Default parameter values are automatically substituted when m, n, d, $m1$, $n1$, and/or $d1$ are given as 0: If $m1 = 0$ or $m1 > 360$, $m1$ is changed to 360; if $n1 = 0$ or $n1 > 250$, $n1$ is changed to 250. Then if m is zero, it is changed to $m1 - m0$; if n is zero, it is changed to $n1 - n0$. If d is zero, it is changed to 255. If $d1$ is zero, it is changed to $255(m1 - m0)(n1 - n0)$. After these substitutions have been made, the parameters must satisfy

$$m0 < m1, \qquad n0 < n1, \qquad \text{and} \qquad d0 < d1.$$

Examples: The call $lisa\_pix = lisa(0,0,0,0,0,0,0,0,0,area)$ is equivalent to the call $lisa\_pix = lisa(360,250,255,0,360,$ $360*250, area)$; this special case delivers the original `lisa.dat` data as a 360×250 array of integers in the range $[0\,..\,255]$. You can access the pixel in row k and column l by writing

$$*(lisa\_pix + n*k + l),$$

where n in this case is 250. A square array extracted from the top part of the picture, leaving out Mona's hands at the bottom, can be obtained by calling $lisa(250,250,255,0,250,0,250,0,0,area)$.

The call $lisa(36,25,25500,0,0,0,0,0,0,area)$ gives a 36×25 array of pixel values in the range $[0\,..\,25500]$, obtained by summing 10×10 subsquares of the original data.

The call $lisa(100,100,100,0,0,0,0,0,0,area)$ gives a 100×100 array of pixel values in the range $[0\,..\,100]$; in this case the original data is effectively broken into subpixels and averaged appropriately. Notice that each output pixel in this example comes from 3.6 input rows and 2.5 input columns; therefore the image is being distorted (compressed vertically). However, our GraphBase applications are generally interested more in combinatorial test data, not in images per se. If $(m1 - m0)/m = (n1 - n0)/n$, the output of *lisa* will represent "square pixels." But if $(m1 - m0)/m < (n1 - n0)/n$, a halftone generated from the output will be compressed in the horizontal dimension; if $(m1 - m0)/m > (n1 - n0)/n$, it will be compressed in the vertical dimension.

If you want to reduce the original image to binary data, with the value 0 wherever the original pixels are less than some threshold value t and the value 1 whenever they are t or more, call $lisa(m,n,1,m0,m1,n0,n1,$ $0, t*(m1 - m0)*(n1 - n0), area)$.

The subroutine call $lisa(1000,1000,255,0,250,0,250,0,0,area)$ produces a million pixels from the upper part of the original image. This matrix contains more entries than the original data in `lisa.dat`, but of course it is not any more accurate; it has simply been obtained by linear interpolation—in fact, by replicating the original data in 4×4 subarrays.

Mona Lisa's famous smile appears in the 16×32 subarray defined by $m0 = 94$, $m1 = 110$, $n0 = 97$, $n1 = 129$. The *smile* macro makes this easily accessible. (See also *eyes*.)

A string *lisa_id* is constructed, showing the actual parameter values used by *lisa* after defaults have been supplied. The *area* parameter is omitted from this string.

⟨ `gb_lisa.h` 1 ⟩ +≡
#define *smile* $m0 = 94, m1 = 110, n0 = 97, n1 = 129$ /* 16×32 */
#define *eyes* $m0 = 61, m1 = 80, n0 = 91, n1 = 140$ /* 20×50 */
extern char *lisa_id*[ ];

4. ⟨ Global variables 4 ⟩ ≡
char *lisa_id*[ ] = `"lisa(360,250,9999999999,359,360,249,250,9999999999,9999999999)"`;

This code is used in section 6.

5. If the *lisa* routine encounters a problem, it returns Λ (`NULL`), after putting a nonzero number into the external variable *panic_code*. This code number identifies the type of failure. Otherwise *lisa* returns a pointer to the newly created array. (The external variable *panic_code* is defined in GB_GRAPH.)

#define *panic*(*c*) { *panic_code* = *c*; *gb_trouble_code* = 0; **return** Λ; }

6. The C file `gb_lisa.c` begins as follows. (Other subroutines come later.)

```
#include "gb_io.h"   /* we will use the GB_IO routines for input */
#include "gb_graph.h"   /* we will use the GB_GRAPH data structures */
  ⟨Preprocessor definitions⟩
  ⟨Global variables 4⟩
  ⟨Private variables 16⟩
  ⟨Private subroutines 15⟩
  long *lisa(m, n, d, m0, m1, n0, n1, d0, d1, area)
      unsigned long m, n;   /* number of rows and columns desired */
      unsigned long d;   /* maximum pixel value desired */
      unsigned long m0, m1;   /* input will be from rows [m0 .. m1) */
      unsigned long n0, n1;   /* and from columns [n0 .. n1) */
      unsigned long d0, d1;   /* lower and upper threshold of raw pixel scores */
      Area area;   /* where to allocate the matrix that will be output */
  { ⟨Local variables for lisa 7⟩
    ⟨Check the parameters and adjust them for defaults 8⟩;
    ⟨Allocate the matrix 9⟩;
    ⟨Read lisa.dat and map it to the desired output form 10⟩;
    return matx;
  }
```

7. ⟨Local variables for *lisa* 7⟩ ≡

```
  long *matx = Λ;   /* the matrix constructed by lisa */
  register long k, l;   /* the current row and column of output */
  register long i, j;   /* all-purpose indices */
  long cap_M, cap_N;   /* m1 − m0 and n1 − n0, dimensions of the input */
  long cap_D;   /* d1 − d0, scale factor */
```

See also sections 11 and 14.

This code is used in section 6.

8. ⟨Check the parameters and adjust them for defaults 8⟩ ≡

```
  if (m1 ≡ 0 ∨ m1 > MAX_M) m1 = MAX_M;
  if (m1 ≤ m0) panic(bad_specs + 1);   /* m0 must be less than m1 */
  if (n1 ≡ 0 ∨ n1 > MAX_N) n1 = MAX_N;
  if (n1 ≤ n0) panic(bad_specs + 2);   /* n0 must be less than n1 */
  cap_M = m1 − m0; cap_N = n1 − n0;
  if (m ≡ 0) m = cap_M;
  if (n ≡ 0) n = cap_N;
  if (d ≡ 0) d = MAX_D;
  if (d1 ≡ 0) d1 = MAX_D * cap_M * cap_N;
  if (d1 ≤ d0) panic(bad_specs + 3);   /* d0 must be less than d1 */
  if (d1 ≥ #80000000) panic(bad_specs + 4);   /* d1 must be less than 2^31 */
  cap_D = d1 − d0;
  sprintf(lisa_id, "lisa(%lu,%lu,%lu,%lu,%lu,%lu,%lu,%lu,%lu)", m, n, d, m0, m1, n0, n1, d0, d1);
```

This code is used in section 6.

9. ⟨Allocate the matrix 9⟩ ≡

```
  matx = gb_typed_alloc(m * n, long, area);
  if (gb_trouble_code) panic(no_room + 1);   /* no room for the output data */
```

This code is used in section 6.

10. ⟨ Read `lisa.dat` and map it to the desired output form 10 ⟩ ≡
⟨ Open the data file, skipping unwanted rows at the beginning 19 ⟩;
⟨ Generate the m rows of output 13 ⟩;
⟨ Close the data file, skipping unwanted rows at the end 20 ⟩;

This code is used in section 6.

11. Elementary image processing. As mentioned in the introduction, we can visualize the input as a giant $mM \times nN$ matrix, into which an $M \times N$ image is placed by replication of pixel values, and from which an $m \times n$ image is derived by summation of pixel values and subsequent scaling. Here $M = m1 - m0$ and $N = n1 - n0$.

Let (κ, λ) be a position in the giant matrix, where $0 \le \kappa < mM$ and $0 \le \lambda < nN$. The corresponding indices of the input image are then $\bigl(m0 + \lfloor \kappa/m \rfloor, n0 + \lfloor \lambda/n \rfloor\bigr)$, and the corresponding indices of the output image are $\bigl(\lfloor \kappa/M \rfloor, \lfloor \lambda/N \rfloor\bigr)$. Our main job is to compute the sum of all pixel values that lie in each given row k and column l of the output image. Many elements are repeated in the sum, so we want to use multiplication instead of simple addition whenever possible.

For example, let's consider the inner loop first, the loop on l and λ. Suppose $n = 3$, and suppose the input pixels in the current row of interest are $\langle a_0, \ldots, a_{N-1} \rangle$. Then if $N = 3$, we want to compute the output pixels $\langle 3a_0, 3a_1, 3a_2 \rangle$; if $N = 4$, we want to compute $\langle 3a_0 + a_1, 2a_1 + 2a_2, a_2 + 3a_3 \rangle$; if $N = 2$, we want to compute $\langle 2a_0, a_0 + a_1, 2a_1 \rangle$. The logic for doing this computation with the proper timing can be expressed conveniently in terms of four local variables:

```
⟨ Local variables for lisa 7 ⟩ +≡
  long *cur_pix;    /* current position within in_row */
  long lambda;    /* right boundary in giant for the input pixel in cur_pix */
  long lam;    /* the first giant column not yet used in the current row */
  long next_lam;    /* right boundary in giant for the output pixel in column l */
```

12. ⟨ Process one row of pixel sums, multiplying them by f 12 ⟩ ≡

```
  lambda = n; cur_pix = in_row + n0;
  for (l = lam = 0; l < n; l++) { register long sum = 0;
    next_lam = lam + cap_N;
    do { register long nl;    /* giant column where something new might happen */
      if (lam ≥ lambda) cur_pix++, lambda += n;
      if (lambda < next_lam) nl = lambda;
      else nl = next_lam;
      sum += (nl − lam) * (*cur_pix);
      lam = nl;
    } while (lam < next_lam);
    *(out_row + l) += f * sum;
  }
```

This code is used in section 13.

13. The outer loop (on k and κ) is similar but slightly more complicated, because it deals with a vector of sums instead of a single sum and because it must invoke the input routine when we're done with a row of input data.

```
⟨ Generate the m rows of output 13 ⟩ ≡
  kappa = 0;
  out_row = matx;
  for (k = kap = 0; k < m; k++) {
    for (l = 0; l < n; l++) *(out_row + l) = 0;    /* clear the vector of sums */
    next_kap = kap + cap_M;
    do { register long nk;    /* giant row where something new might happen */
      if (kap ≥ kappa) {
        ⟨ Read a row of input into in_row 21 ⟩;
        kappa += m;
      }
      if (kappa < next_kap) nk = kappa;
      else nk = next_kap;
      f = nk − kap;
      ⟨ Process one row of pixel sums, multiplying them by f 12 ⟩;
      kap = nk;
    } while (kap < next_kap);
    for (l = 0; l < n; l++, out_row++)    /* note that out_row will advance by n */
      ⟨ Scale the sum found in *out_row 18 ⟩;
  }
```

This code is used in section 10.

14. ⟨ Local variables for *lisa* 7 ⟩ +≡

```
  long kappa;    /* bottom boundary in giant for the input pixels in in_row */
  long kap;    /* the first giant row not yet used */
  long next_kap;    /* bottom boundary in giant for the output pixel in row k */
  long f;    /* factor by which current input sums should be replicated */
  long *out_row;    /* current position in matx */
```

15. Integer scaling. Here's a general-purpose routine to compute $\lfloor na/b \rfloor$ exactly without risking integer overflow, given integers $n \ge 0$ and $0 < a \le b$. The idea is to solve the problem first for $n/2$, if n is too large.

We are careful to precompute values so that integer overflow cannot occur when b is very large.

```
#define el_gordo #7fffffff   /* 2^31 − 1, the largest single-precision long */
⟨ Private subroutines 15 ⟩ ≡
  static long na_over_b(n, a, b)
      long n, a, b;
  { long nmax = el_gordo/a;   /* the largest n such that na doesn't overflow */
    register long r, k, q, br;
    long a_thresh, b_thresh;
    if (n ≤ nmax) return (n ∗ a)/b;
    a_thresh = b − a;
    b_thresh = (b + 1) ≫ 1;   /* ⌈b/2⌉ */
    k = 0;
    do { bit[k] = n & 1;   /* save the least significant bit of n */
      n ≫= 1;   /* and shift it out */
      k++;
    } while (n > nmax);
    r = n ∗ a; q = r/b; r = r − q ∗ b;
    ⟨ Maintain quotient q and remainder r while increasing n back to its original value
        2^k n + (bit[k − 1] . . . bit[0])_2 17 ⟩;
    return q;
  }
```

See also section 32.

This code is used in section 6.

16. ⟨ Private variables 16 ⟩ ≡

```
  static long bit[30];   /* bits shifted out of n */
```

See also section 22.

This code is used in section 6.

17. ⟨ Maintain quotient q and remainder r while increasing n back to its original value $2^k n + (bit[k-1] \ldots bit[0])_2$ 17 ⟩ ≡

```
  do { k−−; q ≪= 1;
    if (r < b_thresh) r ≪= 1;
    else q++, br = (b − r) ≪ 1, r = b − br;
    if (bit[k]) {
      if (r < a_thresh) r += a;
      else q++, r −= a_thresh;
    }
  } while (k);
```

This code is used in section 15.

18. ⟨ Scale the sum found in ∗*out_row* 18 ⟩ ≡

```
  if (∗out_row ≤ d0) ∗out_row = 0;
  else if (∗out_row ≥ d1) ∗out_row = d;
  else ∗out_row = na_over_b(d, ∗out_row − d0, cap_D);
```

This code is used in section 13.

19. Input data format. The file `lisa.dat` contains 360 rows of pixel data, and each row appears on five consecutive lines of the file. The first four lines contain the data for 60 pixels; each sequence of four pixels is represented by five radix-85 digits, using the *icode* mapping of GB_IO. The fifth and final line of each row contains $4 + 4 + 2 = 10$ more pixels, represented as $5 + 5 + 3$ radix-85 digits.

⟨ Open the data file, skipping unwanted rows at the beginning 19 ⟩ ≡

```
if (gb_open("lisa.dat") ≠ 0) panic(early_data_fault);
    /* couldn't open the file; io_errors tells why */
for (i = 0; i < m0; i++)
  for (j = 0; j < 5; j++) gb_newline();    /* ignore one row of data */
```

This code is used in section 10.

20. ⟨ Close the data file, skipping unwanted rows at the end 20 ⟩ ≡

```
for (i = m1; i < MAX_M; i++)
  for (j = 0; j < 5; j++) gb_newline();    /* ignore one row of data */
if (gb_close() ≠ 0) panic(late_data_fault);    /* checksum or other failure in data file; see io_errors */
```

This code is used in section 10.

21. ⟨ Read a row of input into *in_row* 21 ⟩ ≡

```
{ register long dd;
  for (j = 15, cur_pix = &in_row[0]; ; cur_pix += 4) {
    dd = gb_digit(85); dd = dd * 85 + gb_digit(85); dd = dd * 85 + gb_digit(85);
    if (cur_pix ≡ &in_row[MAX_N − 2]) break;
    dd = dd * 85 + gb_digit(85); dd = dd * 85 + gb_digit(85);
    *(cur_pix + 3) = dd & #ff; dd = (dd ≫ 8) & #ffffff;
    *(cur_pix + 2) = dd & #ff; dd ≫= 8;
    *(cur_pix + 1) = dd & #ff; *cur_pix = dd ≫ 8;
    if (−−j ≡ 0) gb_newline(), j = 15;
  }
  *(cur_pix + 1) = dd & #ff; *cur_pix = dd ≫ 8; gb_newline();
}
```

This code is used in section 13.

22. ⟨ Private variables 16 ⟩ +≡

```
static long in_row[MAX_N];
```

23. Planar graphs. We can obtain a large family of planar graphs based on digitizations of Mona Lisa by using the following simple scheme: Each matrix of pixels defines a set of connected regions containing pixels of the same value. (Two pixels are considered adjacent if they share an edge.) These connected regions are taken to be vertices of an undirected graph; two vertices are adjacent if the corresponding regions have at least one pixel edge in common.

We can also state the construction another way. If we take any planar graph and collapse two adjacent vertices, we obtain another planar graph. Suppose we start with the planar graph having mn vertices $[k,l]$ for $0 \le k < m$ and $0 \le l < n$, where $[k,l]$ is adjacent to $[k,l-1]$ when $l > 0$ and to $[k-1,l]$ when $k > 0$. Then we can attach pixel values to each vertex, after which we can repeatedly collapse adjacent vertices whose pixel values are equal. The resulting planar graph is the same as the graph of connected regions that was described in the previous paragraph.

The subroutine call *plane_lisa*($m, n, d, m0, m1, n0, n1, d0, d1$) constructs the planar graph associated with the digitization produced by *lisa*. The description of *lisa*, given earlier, explains the significance of parameters m, n, d, $m0$, $m1$, $n0$, $n1$, $d0$, and $d1$. There will be at most mn vertices, and the graph will be simply an $m \times n$ grid unless d is small enough to permit adjacent pixels to have equal values. The graph will also become rather trivial if d is too small.

Utility fields *first_pixel* and *last_pixel* give, for each vertex, numbers of the form $k * n + l$, identifying the topmost/leftmost and bottommost/rightmost positions $[k,l]$ in the region corresponding to that vertex. Utility fields *matrix_rows* and *matrix_cols* in the **Graph** record contain the values of m and n; thus, in particular, the value of n needed to decompose *first_pixel* and *last_pixel* into individual coordinates can be found in *g→matrix_cols*.

The original pixel value of a vertex is placed into its *pixel_value* utility field.

```
#define pixel_value  x.I
#define first_pixel  y.I
#define last_pixel  z.I
#define matrix_rows  uu.I
#define matrix_cols  vv.I

  Graph *plane_lisa(m, n, d, m0, m1, n0, n1, d0, d1)
       unsigned long m, n;    /* number of rows and columns desired */
       unsigned long d;    /* maximum value desired */
       unsigned long m0, m1;    /* input will be from rows [m0 .. m1) */
       unsigned long n0, n1;    /* and from columns [n0 .. n1) */
       unsigned long d0, d1;    /* lower and upper threshold of raw pixel scores */
  { ⟨Local variables for plane_lisa 24⟩
    init_area(working_storage);
    ⟨Figure out the number of connected regions, regs 26⟩;
    ⟨Set up a graph with regs vertices 29⟩;
    ⟨Put the appropriate edges into the graph 30⟩;
  trouble: gb_free(working_storage);
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);    /* oops, we ran out of memory somewhere back there */
    }
    return new_graph;
  }
```

24. ⟨ Local variables for *plane_lisa* 24 ⟩ ≡

```
Graph *new_graph;    /* the graph constructed by plane_lisa */
register long j, k, l;    /* all-purpose indices */
Area working_storage;    /* tables needed while plane_lisa does its thinking */
long *a;    /* the matrix constructed by lisa */
long regs = 0;    /* number of vertices generated so far */
```

See also sections 27 and 31.

This code is used in section 23.

25. ⟨ gb_lisa.h 1 ⟩ +≡

```
#define pixel_value x.I    /* definitions for the header file */
#define first_pixel y.I
#define last_pixel z.I
#define matrix_rows uu.I
#define matrix_cols vv.I
```

26. The following algorithm for counting the connected regions considers the array elements $a[k,l]$ to be linearly ordered as they appear in memory. Thus we can speak of the n elements preceding a given element $a[k,l]$, if $k>0$; these are the elements $a[k,l-1]$, …, $a[k,0]$, $a[k-1,n-1]$, …, $a[k-1,l]$. These n elements appear in n different columns.

During the algorithm, we move through the array from bottom right to top left, maintaining an auxiliary table $\langle f[0],\ldots,f[n-1]\rangle$ with the following significance: Whenever two of the n elements preceding our current position $[k,l]$ are connected to each other by a sequence of pixels with equal value, where the connecting links do not involve pixels more than n steps before our current position, those elements will be linked together in the f array. More precisely, we will have $f[c_1]=c_2$, …, $f[c_{j-1}]=c_j$, and $f[c_j]=c_j$, when there are j equivalent elements in columns c_1, …, c_j. Here c_1 will be the "last" column and c_j the "first," in wraparound order; each element with $f[c]\neq c$ points to an earlier element.

The main function of the f table is to identify the topmost/leftmost pixel of a region. If we are at position $[k,l]$ and if we find $f[l]=l$ while $a[k-1,l]\neq a[k,l]$, there is no way to connect $[k,l]$ to earlier positions, so we create a new vertex for it.

We also change the a matrix, to facilitate another algorithm below. If position $[k,l]$ is the topmost/leftmost pixel of a region, we set $a[k,l]=-1-a[k,l]$; otherwise we set $a[k,l]=f[l]$, the column of a preceding element belonging to the same region.

⟨ Figure out the number of connected regions, *regs* 26 ⟩ ≡

```
a = lisa(m, n, d, m0, m1, n0, n1, d0, d1, working_storage);
if (a ≡ Λ) return Λ;    /* panic_code has been set by lisa */
sscanf(lisa_id, "lisa(%lu,%lu,", &m, &n);    /* adjust for defaults */
f = gb_typed_alloc(n, unsigned long, working_storage);
if (f ≡ Λ) {
  gb_free(working_storage);    /* recycle the a matrix */
  panic(no_room + 2);    /* there's no room for the f vector */
}
⟨ Pass over the a matrix from bottom right to top left, looking for the beginnings of connected regions 28 ⟩;
```

This code is used in section 23.

27. ⟨ Local variables for *plane_lisa* 24 ⟩ +≡

```
unsigned long *f;    /* beginning of array f; f[j] is the column of an equivalent element */
long *apos;    /* the location of a[k,l] */
```

28. We maintain a pointer *apos* equal to $\&a[k,l]$, so that $*(apos-1)=a[k,l-1]$ and $*(apos-n)=a[k-1,l]$ when $l>0$ and $k>0$.

The loop that replaces $f[j]$ by j can cause this algorithm to take time mn^2. We could improve the worst case by using path compression, but the extra complication is rarely worth the trouble.

⟨ Pass over the a matrix from bottom right to top left, looking for the beginnings of connected regions 28 ⟩ ≡

```
for (k = m, apos = a + n * (m + 1) - 1; k ≥ 0; k--)
  for (l = n - 1; l ≥ 0; l--, apos--) {
    if (k < m) {
      if (k > 0 ∧ *(apos - n) ≡ *apos) {
        for (j = l; f[j] ≠ j; j = f[j]) ;   /* find the first element */
        f[j] = l;   /* link it to the new first element */
        *apos = l;
      } else if (f[l] ≡ l) *apos = -1 - *apos, regs++;   /* new region found */
      else *apos = f[l];
    }
    if (k > 0 ∧ l < n - 1 ∧ *(apos - n) ≡ *(apos - n + 1)) f[l + 1] = l;
    f[l] = l;
  }
```

This code is used in section 26.

29. ⟨ Set up a graph with *regs* vertices 29 ⟩ ≡

```
new_graph = gb_new_graph(regs);
if (new_graph ≡ Λ) panic(no_room);   /* out of memory before we're even started */
sprintf(new_graph→id, "plane_%s", lisa_id);
strcpy(new_graph→util_types, "ZZZIIIZZIIZZZZ");
new_graph→matrix_rows = m;
new_graph→matrix_cols = n;
```

This code is used in section 23.

30. Now we make another pass over the matrix, this time from top left to bottom right. An auxiliary vector of length n is once again sufficient to tell us when one region is adjacent to a previous one. In this case the vector is called u, and it contains pointers to the vertices in the n positions before our current position. We assume that a pointer to a **Vertex** takes the same amount of memory as an **unsigned long**, hence u can share the space formerly occupied by f; if this is not the case, a system-dependent change should be made here.

The vertex names are simply integers, starting with 0.

⟨ Put the appropriate edges into the graph 30 ⟩ ≡

```
regs = 0;
u = (Vertex **) f;
for (l = 0; l < n; l++) u[l] = Λ;
for (k = 0, apos = a, aloc = 0; k < m; k++)
  for (l = 0; l < n; l++, apos++, aloc++) {
    w = u[l];
    if (*apos < 0) {
      sprintf(str_buf, "%ld", regs);
      v = new_graph→vertices + regs;
      v→name = gb_save_string(str_buf);
      v→pixel_value = −*apos − 1;
      v→first_pixel = aloc;
      regs++;
    } else v = u[*apos];
    u[l] = v;
    v→last_pixel = aloc;
    if (gb_trouble_code) goto trouble;
    if (k > 0 ∧ v ≠ w) adjac(v, w);
    if (l > 0 ∧ v ≠ u[l − 1]) adjac(v, u[l − 1]);
  }
```

This code is used in section 23.

31. ⟨ Local variables for *plane_lisa* 24 ⟩ +≡

```
Vertex **u;  /* table of vertices for previous n pixels */
Vertex *v;  /* vertex corresponding to position [k, l] */
Vertex *w;  /* vertex corresponding to position [k − 1, l] */
long aloc;  /* k * n + l */
```

32. The *adjac* routine makes two vertices adjacent, if they aren't already. A faster way to recognize duplicates would probably speed things up.

⟨ Private subroutines 15 ⟩ +≡

```
static void adjac(u, v)
    Vertex *u, *v;
{ Arc *a;
  for (a = u→arcs; a; a = a→next)
    if (a→tip ≡ v) return;
  gb_new_edge(u, v, 1L);
}
```

33. Bipartite graphs. An even simpler class of Mona-Lisa-based graphs is obtained by considering the m rows and n columns to be individual vertices, with a row adjacent to a column if the associated pixel value is sufficiently large or sufficiently small. All edges have length 1.

The subroutine call *bi_lisa*$(m, n, m0, m1, n0, n1, thresh, c)$ constructs the bipartite graph corresponding to the $m \times n$ digitization produced by *lisa*, using parameters $(m0, m1, n0, n1)$ to define a rectangular subpicture as described earlier. The threshold parameter *thresh* should be between 0 and 65535. If the pixel value in row k and column l is at least $thresh/65535$ of its maximum, vertices k and l will be adjacent. If $c \neq 0$, however, the convention is reversed; vertices are then adjacent when the corresponding pixel value is *smaller* than $thresh/65535$. Thus adjacencies come from "light" areas of da Vinci's painting when $c = 0$ and from "dark" areas when $c \neq 0$. There are $m + n$ vertices and up to $m \times n$ edges.

The actual pixel value is recorded in utility field $b.I$ of each arc, and scaled to be in the range $[0, 65535]$.

```
Graph *bi_lisa(m, n, m0, m1, n0, n1, thresh, c)
    unsigned long m, n;   /* number of rows and columns desired */
    unsigned long m0, m1;   /* input will be from rows [m0 .. m1) */
    unsigned long n0, n1;   /* and from columns [n0 .. n1) */
    unsigned long thresh;   /* threshold defining adjacency */
    long c;   /* should we prefer dark pixels to light pixels? */
{ ⟨Local variables for bi_lisa 34⟩
  init_area(working_storage);
  ⟨Set up a bipartite graph with m + n vertices 35⟩;
  ⟨Put the appropriate edges into the bigraph 36⟩;
  gb_free(working_storage);
  if (gb_trouble_code) {
    gb_recycle(new_graph);
    panic(alloc_fault);   /* oops, we ran out of memory somewhere back there */
  }
  return new_graph;
}
```

34. ⟨Local variables for *bi_lisa* 34⟩ ≡

```
Graph *new_graph;   /* the graph constructed by bi_lisa */
register long k, l;   /* all-purpose indices */
Area working_storage;   /* tables needed while bi_lisa does its thinking */
long *a;   /* the matrix constructed by lisa */
long *apos;   /* the location of a[k, l] */
register Vertex *u, *v;   /* current vertices of interest */
```

This code is used in section 33.

35. ⟨ Set up a bipartite graph with $m + n$ vertices 35 ⟩ ≡

```
a = lisa(m, n, 65535L, m0, m1, n0, n1, 0L, 0L, working_storage);
if (a ≡ Λ) return Λ;   /* panic_code has been set by lisa */
sscanf(lisa_id, "lisa(%lu,%lu,65535,%lu,%lu,%lu,%lu", &m, &n, &m0, &m1, &n0, &n1);
new_graph = gb_new_graph(m + n);
if (new_graph ≡ Λ) panic(no_room);   /* out of memory before we're even started */
sprintf(new_graph→id, "bi_lisa(%lu,%lu,%lu,%lu,%lu,%lu,%lu,%c)", m, n, m0, m1, n0, n1, thresh,
      c ? '1' : '0');
new_graph→util_types[7] = 'I';   /* enable field b.I */
mark_bipartite(new_graph, m);
for (k = 0, v = new_graph→vertices; k < m; k++, v++) {
   sprintf(str_buf, "r%ld", k);   /* row vertices are called "r0", "r1", etc. */
   v→name = gb_save_string(str_buf);
}
for (l = 0; l < n; l++, v++) {
   sprintf(str_buf, "c%ld", l);   /* column vertices are called "c0", "c1", etc. */
   v→name = gb_save_string(str_buf);
}
```

This code is used in section 33.

36. Since we've called *lisa* with $d = 65535$, the determination of adjacency is simple.

⟨ Put the appropriate edges into the bigraph 36 ⟩ ≡

```
for (u = new_graph→vertices, apos = a; u < new_graph→vertices + m; u++)
   for (v = new_graph→vertices + m; v < new_graph→vertices + m + n; apos++, v++) {
      if (c ? *apos < thresh : *apos ≥ thresh) {
         gb_new_edge(u, v, 1L);
         u→arcs→b.I = v→arcs→b.I = *apos;
      }
   }
```

This code is used in section 33.

37. Index. As usual, we close with an index that shows where the identifiers of *gb_lisa* are defined and used.

GB_LISA

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