July 4, 2016 at 06:31

1. Introduction. I'm trying to calculate a few billion Ulam numbers. This sequence

$$(U_1, U_2, \ldots) = (1, 2, 3, 4, 6, 8, 11, 13, 16, 18, 26, 28, 36, 38, 47, 48, 53, 57, 62, 69, \ldots)$$

is defined by setting $U_1 = 1$, $U_2 = 2$, and thereafter letting U_{n+1} be the smallest number greater than U_n that can be written $U_j + U_k$ for exactly one pair (j, k) with $1 \le j < k \le n$. (Such a number must exist; otherwise the pair $(j, k) = (n-1, n)$ would qualify and lead to a contradiction.)

The related sequence

$$(1, 2, 23, 25, 33, 35, 43, 45, 67, 92, 94, 96, 111, 121, 136, \ldots)$$

of "Ulam misses" contains all numbers that cannot be expressed as the sum of two distinct Ulams.

This program is based on some beautiful ideas due to Philip E. Gibbs, whose Java code in 2015 was first to beat the billion-number barrier. It runs much, much faster than the bitwise-oriented program ULAM that I wrote ten years ago. And it has some interesting touches that taught me some lessons, which I'm keen to pass on to others.

Ulam mentioned this sequence in *SIAM Review* **6** (1964), 348, as part of a more general discussion. Its properties have baffled number theorists for many years; but new insights are beginning to change the picture: Stefan Steinerberger discovered empirically that $U_n/\lambda \bmod 1$ almost always lies in the interval $[\frac{1}{3} \,.\,. \frac{2}{3}]$, where $\lambda \approx 2.443443$ ["A hidden signal in the Ulam sequence," Report DCS/TR-1508 (Yale University, 2015)]. Then Gibbs ["An efficient method for computing Ulam numbers," viXra:1508.0085 (2015)] exploited that property in nontrivial ways, finding that roughly $O(N)$ time and $O(N)$ space suffice to compute the first N terms. He subsequently discovered how to significantly decrease the coefficients of N in the time and space requirements; and when I asked him how he did it, he kindly sent me a copy of his program.

Of course I couldn't resist translating it from Java into `CWEB`, because that's what I do for a living. So this is the result.

2. This program has lots of tunable parameters, and it should prove to be interesting to see how they affect the performance. Of course the main parameter is N, the desired number of outputs. Other options are preceded on the command line by a letter; for example, '`v5`' sets the verboseness parameter to 5.

Each parameter will be explained later, but it's convenient to summarize the option letters here:

- '`v`⟨integer⟩' to enable various binary-coded levels of verbose output on *stderr* (default=1).
- '`p`⟨positive integer⟩' to specify the numerator of a rational approximation to λ (default=120500181).
- '`q`⟨positive integer⟩' to specify the denominator of a rational approximation to λ (default=49315733). The program assumes that p and q are less than 2^{32}, and that $2 < p/q \le 3$.
- '`m`⟨positive integer⟩' to specify the spacing of outputs; every mth Ulam number will be written to standard output. (The default is $m = 1000000$; `m0` will report only U_N.)
- '`g`⟨positive integer⟩' to specify the largest gap for which statistics are kept (default=2000).
- '`o`⟨positive integer⟩' to specify the space allocated for "outliers" and "near-outliers" (default=1000000).
- '`i`⟨positive integer⟩' to specify the size of the indexes to those lists (default=100000).
- '`T`⟨positive real⟩' to specify the threshold in the definition of 'near outlier' (default=100).
- '`b`⟨positive integer⟩' to specify the number of bits of the *is_um* table that are stored in a single byte (default=18). (That default is optimum: `b19` turns out to be too high, if $N > 2198412$.)
- '`B`⟨positive integer⟩' to specify the number of initial *is_ulam* entries that are encoded with one bit per byte (default=18000). This value should be a multiple of the `b` option, and at least 3.
- '`w`⟨positive integer⟩' to specify the window size for remembering recently computed Ulam numbers (default=1000000). The window size must be at least 3.
- '`M`⟨filename⟩' to produce METAPOST illustrations showing the distributions of Ulam numbers and Ulam misses, modulo λ.

3. The *vbose* parameter is the sum of the following binary codes. To enable everything, you can say '`v-1`'.

```
#define show_usage_stats 1 /* reports time and space usage */
#define show_compression_stats 2 /* reports details of is_ulam encoding */
#define show_histograms 4 /* reports Ulams and misses mod λ */
#define show_gap_stats 8 /* gives histogram and examples of every gap */
#define show_record_gaps 16 /* reports every gap that exceeded all precedessors */
#define show_record_outliers 32 /* reports outliers that exceeded earlier ones */
#define show_outlier_details 64 /* reports insertion or deletion of all outliers */
#define show_record_cutoffs 128 /* reports residue cutoffs for near outliers */
#define show_omitted_inliers 256 /* reports inliers that aren't near outliers */
#define show_brute_winners 512 /* reports unusual cases after brute-force trials */
#define show_inlier_anchors 1024 /* reports cases when two inliers make Ulam */
```

4. Here then is an outline of the whole program:

```
#define o mems++ /* count one mem (one access to or from 64 bits of memory) */
#define oo mems += 2 /* count two mems */
#define ooo mems += 3 /* count three mems */
#define O "%" /* used for percent signs in format strings */
#define mod % /* used for percent signs denoting remainder in C */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
  typedef unsigned char uchar; /* a convenient abbreviation */
  typedef unsigned int uint; /* ditto */
  typedef unsigned long long ullng; /* ditto */
  ⟨Type definitions 9⟩
  ⟨Global variables 6⟩
  ⟨Subroutines 10⟩
  main(int argc, char *argv[])
  {
    register int i, j, k, r, rp, t, x, y, hits, count;
    register ullng n, u, up;
    ⟨Process the command line 5⟩;
    ⟨Allocate the arrays 16⟩;
    ⟨Initialize the data structures 17⟩;
    for (u = 3; n < maxn; u++) ⟨Decide whether u is an Ulam number or an Ulam miss or neither, and
          update the data structures accordingly 32⟩;
    if (mp_file) ⟨Output the METAPOST file 56⟩;
  finish_up: ⟨Print farewell messages 51⟩;
  }
```

5. If a command-line parameter is specified twice, the first one wins.

⟨ Process the command line 5 ⟩ ≡
```
if (argc ≡ 1) k = 1;
else {
  k = sscanf(argv[argc − 1], ""O"lld", &maxn) − 1;   /* read N */
  for (j = argc − 2; j; j−−)
    switch (argv[j][0]) {
      ⟨ Respond to a command-line option, setting k nonzero on error 7 ⟩;
    default: k = 1;   /* unrecognized command-line option */
    }
}
⟨ If there's a problem, print a message about Usage: and exit 8 ⟩;
```
This code is used in section 4.

6. ⟨ Global variables 6 ⟩ ≡
```
ullng maxn;   /* desired number of Ulams to compute */
int vbose = show_usage_stats;   /* level of verbosity */
uint lamp = 120500181;   /* numerator of λ */
uint lamq = 49315733;   /* denominator of λ */
ullng spacing;   /* spacing between outputs; 0 means give only the last */
ullng misses;   /* we've seen these many Ulam misses */
int biggestgap = 1;   /* the largest gap seen so far */
int maxgap = 2000;   /* the largest gap for which we keep histogram data */
int outliers = 1000000;   /* maximum number of outliers and near-outliers to remember */
int isize = 100000;   /* total size of the two indexes (is always even) */
double thresh = 100;   /* theshold for remembering a near-outlier */
ullng mems, last_mems;   /* mem count */
clock_t last_clock;   /* the last time we called clock() */
ullng bytes;   /* memory used by main data structures */
int bits_per_compressed_byte = 18;   /* packing parameter */
int uncompressed_bytes = 18000;   /* this many initial is_ulam bits not packed */
ullng window_size = 1000000;   /* we remember this many previous Ulams */
FILE *mp_file;   /* file for optional output of METAPOST code */
char *mp_name;   /* its name */
```
See also sections 15, 25, 35, 45, and 57.

This code is used in section 4.

7. ⟨ Respond to a command-line option, setting k nonzero on error 7 ⟩ ≡

```
case 'v': k |= (sscanf(argv[j] + 1, ""O"d", &vbose) − 1); break;
case 'p': k |= (sscanf(argv[j] + 1, ""O"u", &lamp) − 1); break;
case 'q': k |= (sscanf(argv[j] + 1, ""O"u", &lamq) − 1); break;
case 'm': k |= (sscanf(argv[j] + 1, ""O"lld", &spacing) − 1); break;
case 'g': k |= (sscanf(argv[j] + 1, ""O"d", &maxgap) − 1); break;
case 'o': k |= (sscanf(argv[j] + 1, ""O"d", &outliers) − 1); break;
case 'i': k |= (sscanf(argv[j] + 1, ""O"d", &isize) − 1);
  isize = (isize + 1) & −2; break;    /* round isize up to nearest even number */
case 'T': k |= (sscanf(argv[j] + 1, ""O"lg", &thresh) − 1); break;
case 'b': k |= (sscanf(argv[j] + 1, ""O"d", &bits_per_compressed_byte) − 1); break;
case 'B': k |= (sscanf(argv[j] + 1, ""O"d", &uncompressed_bytes) − 1); break;
case 'w': k |= (sscanf(argv[j] + 1, ""O"lld", &window_size) − 1); break;
case 'M': mp_name = argv[j] + 1, mp_file = fopen(mp_name, "w");
  if (¬mp_file) fprintf(stderr, "Sorry,␣I␣can't␣open␣file␣`"O"s'␣for␣writing!\n", mp_name);
  break;
```

This code is used in section 5.

8. ⟨ If there's a problem, print a message about `Usage:` and *exit* 8 ⟩ ≡

```
  if (k ∨ uncompressed_bytes < 3 ∨ uncompressed_bytes mod bits_per_compressed_byte ∨
        (lamp − 1)/lamq ≠ 2 ∨ window_size < 3) {
    fprintf(stderr, "Usage:␣"O"s␣[v<n>]␣[p<n>]␣[q<n>]␣[m<n>]␣[g<n>]␣[o<n>]␣[i<n>]", argv[0]);
    fprintf(stderr, "␣[T<f>]␣[b<n>]␣[B<n>]␣[w<n>]␣[Mfoo.mp]␣N\n");
    exit(−1);
  }
```

This code is used in section 5.

9. Statistics about important loop counts are kept in **stat** structures.

⟨ Type definitions 9 ⟩ ≡

```
  typedef struct {
    ullng n;    /* the number of samples */
    float mean;    /* the empirical mean */
    int max;    /* the empirical maximum */
    ullng ex;    /* the extreme example that led to max */
  } stat;
```

See also section 24.

This code is used in section 4.

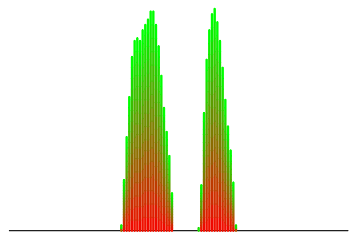
10. ⟨ Subroutines 10 ⟩ ≡

```
  void record_stat(stat *s, int datum, ullng u) {
    if (s→n ≡ 0) s→n = 1, s→mean = (float) datum, s→max = datum, s→ex = u;
    else {
      s→n++;
      s→mean += ((float) datum − s→mean)/((float) s→n);
      if (datum > s→max) s→max = datum, s→ex = u;
    }
  }
```

See also sections 18, 28, 29, and 30.

This code is used in section 4.

11. The ideas behind the algorithm. Gibbs's method is based on the amazing fact that almost all of the values $(U_n/\lambda) \bmod 1$ lie between $1/3$ and $2/3$. Indeed, here's one of the pictures produced by the METAPOST option of this program, showing the distribution of those residues for $1 \le n \le N = 1000000$:



The colors range from green for small n to red for n near N, so we can see the way things "settle down" to a fairly stable distribution as n grows.

Let U be an integer, and let $\rho = (U/\lambda) \bmod 1$ be its associated residue. We might as well assume that the quasi-period length λ is irrational, since "God wouldn't have wanted a rational number that occurs in problems like this to have a really big denominator." Under that assumption, ρ is never a rational number, and $\rho \ne \rho'$ when $U \ne U'$. (Of course, we will actually do our calculations using a rational approximation to λ; hence we'll run into many cases where $\rho = \rho'$.)

Steinerberger found empirically in 2015 that ρ_n lies between $1/4$ and $3/4$ for all known values of U_n, except for four cases: $U_2 = 2$, $\rho_2 \approx .82$; $U_3 = 3$, $\rho_2 \approx .23$; $U_{15} = 47$, $\rho_{15} \approx .23$; $U_{20} = 69$, $\rho_{20} \approx .24$. The reasons for this are unclear, but the facts speak for themselves.

Gibbs went further and defined U to be an 'outlier' if $\rho < 1/3$ or $\rho > 2/3$. He observed that there must be infinitely many outliers, because the sum of two 'inliers' cannot be an 'inlier'. But he conjectured that, for any $\epsilon > 0$, there are only finitely many n with $\rho_n < 1/3 - \epsilon$ or $\rho_n > 2/3 + \epsilon$. And he observed that in the vast majority of known cases, the unique representation $U_n = U_i + U_j$ has the property that either U_i or U_j is an outlier.

Let's pursue this further. If $U = U' + U''$, then we have either $\rho = \rho' + \rho''$ or $\rho = \rho' + \rho'' - 1$. The second case can be written $\bar\rho = \bar\rho' + \bar\rho''$, where $\bar\rho = 1 - \rho$.

If $\rho < 1/4$ or $\rho > 3/4$, it turns out that we can almost always find two completely different representations of U as a sum of two Ulam numbers, using a short brute-force search.

On the other hand, if $1/4 < \rho < 3/4$, we can usually decide whether U is a sum of Ulam numbers $U' + U''$ by looking at relatively few cases where $\rho = \rho' + \rho''$ and $\rho' < \rho''$ or $\bar\rho = \bar\rho' + \bar\rho''$ and $\bar\rho' < \bar\rho''$. Gibbs discovered empirically that it suffices to try cases where U' is either an outlier or a 'near outlier', where the latter is defined by the condition

$$\rho' < 1/2 \text{ and } (\rho' - 1/3)\sqrt{U'} \le \theta \qquad \text{or} \qquad \bar\rho' < 1/2 \text{ and } (\bar\rho' - 1/3)\sqrt{U'} \le \theta$$

and θ is the thresh parameter *thresh* in our program. If U' is large and $\rho' > 1/3$, we won't need to consider U' unless ρ' is *extremely* close to $1/3$.

Consequently we needn't remember detailed information about too many of the Ulam numbers already computed. The brute-force search requires only a reasonably small window; the other searches require only a dictionary of outliers and near-outliers U', sorted by ρ'.

12. Besides those relatively short tables, we also need a way to determine whether or not a given number $u \le U_N$ is an Ulam number. It's known empirically that $U_N \approx 13.5178N$, with minor fluctuations; thus we can safely assume that $U_N < 14N$, and a table of $14N$ bits will suffice.

Still, $14N$ bits is $1.75N$ bytes, which can be substantial when N is many billions. Gibbs was working with just 16 gigabytes of memory, and necessity was the mother of invention: He devised a way to reduce this storage requirement to only $.778N$ bytes, by packing 18 bits into a single byte. This reduction turned out to be possible, and even convenient, because the bit patterns have somewhat low entropy. In fact, at most 256 different patterns actually occur in the *is_ulam* table for 18 consecutive values of n, provided that n is large enough to make the quasi-periodic system relatively stable.

13. Gibbs's early program used floating-point arithmetic to compute the residues ρ. But that led to tricky cases and subtle problems. Then he realized that rational approximations to λ are able to avoid rounding errors, and his program became simpler besides.

He found a good approximation to λ empirically, by adjusting it until the number of "low" outliers with $\rho < 1/3$ was essentially equal to the number of "high" outliers with $\rho > 2/3$. This value was

$$\lambda \approx 2.443442967784743,$$

with the next digits as yet undetermined. Consequently the regular continued fraction is

$$\lambda = 2 + /\!/2, 3, 1, 11, 1, 1, 4, 1, 1, 7, 1, 2, 1, 1, 2, 2, 1, 3, 1, 2, \ldots /\!/,$$

using the notation of *Seminumerical Algorithms*, §4.5.3. Truncating this continued fraction gives good rational approximations to λ; in fact they're the "best possible" such approximations, according to the theorem of Lagrange in exercise 4.5.3–42:

$$2;\quad \frac{5}{2};\quad \frac{17}{7};\quad \frac{22}{9};\quad \frac{259}{106};\quad \frac{281}{115};\quad \frac{540}{221};\quad \frac{2441}{999};\quad \ldots;\quad \frac{35876494}{14682763};\quad \frac{84623687}{34632970} \text{ or } \frac{120500181}{49315733}.$$

The latter two seem to bracket the true value of λ. The final one is the current default, but the other one will probably give equally good results.

When we use the approximation $\lambda = p/q$, the formula $\rho = U/\lambda \bmod 1$ becomes transformed:

$$r = qU \bmod p.$$

The residue is now an *integer* called r, and it lies between 0 and $p-1$, instead of being a fraction ρ between 0 and 1. (Program variables *lamp* and *lamq* correspond to p and q.)

14. These ideas may be easiest to absorb if we work first with small numbers. Suppose $p = 22$ and $q = 9$; this gives a fairly decent approximation $2.4444\ldots$ to λ. The first 100 values of $r_n = 9U_n \bmod 22$ turn out to be nicely concentrated:

$$\{5,5,5,5,6,6,6,6,6,6,6,6,6,7,7,7,7,7,7,7,7,7,7,7,8,8,8,8,8,8,8,8,8,8,8,8,8,9,9,9,9,9,9,9,9,\\ 10,10,10,10,10,10,10,10,11,11,11,11,11,11,11,11,11,11,11,11,12,12,12,12,12,12,12,12,\\ 12,12,12,12,12,12,12,13,13,13,13,13,13,13,14,14,14,14,14,14,14,15,15,15,15,16,16,18\}.$$

Using the better approximation $\lambda \approx 540/221 = 2.44344\ldots$, $r_n = 221U_n \bmod 540$ gives more detail:

$$\{123,127,129,148,166,173,176,177,182,185,185,189,198,202,202,204,206,206,208,209,\\ 210,211,217,218,220,221,222,225,227,230,233,234,235,237,241,242,243,244,246,246,\\ 248248,249,252,252,258,261,262,265,271,277,278,279,282,289,293,296,298,299,301,\\ 302,303,306,308,308,309,311,316,318,324,325,327,327,330,331,332,334,335,336,337,\\ 339,341,342,344,344,346,346,348,354,360,363,373,376,377,380,393,396,399,402,442\}.$$

The outliers for $\lambda = 540/221$ have $r < 180$ or $r \ge 360$. Note that $U_{100} = 690$.

15. The compression scheme. Let's build up some confidence by beginning to write low-level routines for the *is_ulam* table. That table consists of two parts: For $0 \le n < \mathit{uncompressed\_bytes}$, we simply have $\mathit{is\_ulam}[n] = 1$ when n is an Ulam number, $\mathit{is\_ulam}[n] = 0$ when it isn't. But for $n \ge \mathit{uncompressed\_bytes}$, a compressed table called *is_um* contains the necessary information in a lightly encoded form.

Namely, let $b = \mathit{bits\_per\_compressed\_byte}$ be the `b` option on the command line (normally 18). Then $\mathit{is\_um}[n/b]$ will be a byte t such that $\mathit{is\_ulam}[n]$ appears as bit $n \bmod b$ of $\mathit{code}[t]$. This convention applies for $\mathit{uncompressed\_bytes} \le n < \mathit{cur\_slot}$, where *cur_slot* is $b \times \lfloor u/b \rfloor$ and u is the number that we're currently examining. Finally, the *is_ulam* bits for b numbers beginning at *cur_slot* are maintained as the b-bit number *cur_code*.

Of course we must give up if more than 256 different codewords are needed. Auxiliary tables are maintained to provide further information: $\mathit{code\_use}[t]$ records the number of times we've used $\mathit{code}[t]$; $\mathit{code\_example}[t]$ records the smallest *cur_slot* that needed $\mathit{code}[t]$. Such information is maintained behind the scenes, although I could have omitted *code_use* and *code_example* if I were going all out for speed. Their values are always calculated, but reported only if *show_compression_stats* is selected.

The *is_um* table accounts for most of the memory required by this program. It occupies $\lceil 14\mathit{maxn}/b \rceil$ bytes, because $14\mathit{maxn}$ is an upper bound on the numbers u that we need to consider. (Notice that the first $\mathit{uncompressed\_bytes}/b$ of *is_um* are never used. That's a small price to pay for ease of programming.)

⟨ Global variables 6 ⟩ +≡

```
uchar *is_ulam, *is_um;   /* the main arrays for ulamness tests */
ullng cur_sl;   /* this many bytes of is_um have been set correctly */
ullng cur_slot;   /* bits_per_compressed_byte * cur_sl */
uint cur_code = 0;   /* the next bits_per_compressed_byte bits to be compressed */
uint code[256];   /* the expanded "meaning" of each compressed byte */
uchar *inv_code;   /* inverse of the code table */
int code_ptr = 1;   /* this many codes have been defined so far */
ullng code_use[256], code_example[256];   /* the code stats */
```

16. Full disclosure: The number of memory bytes used, kept in *bytes*, accounts only for necessary tables like *is_ulam*, *is_um*, and *code*. It doesn't mention the memory that is devoted to diagnostic data, in arrays such as *code_use* or *code_example*. Any memory allocated to the program itself, and to its atomic global variables, is also blithely ignored.

I also ignore the cost of system calls to *malloc* and *calloc*; the memory accesses that they make, while this program is launching itself, are not reported in *mems*.

```
#define alloc_quit(name, size)
        {
          fprintf(stderr, "Couldn't allocate the "O"s array (size "O"lld)!\n",
               name, (long long) size);
          exit(−666);
        }
```

⟨ Allocate the arrays 16 ⟩ ≡

```
  is_ulam = (uchar ∗) malloc(uncompressed_bytes ∗ sizeof(uchar));
  if (¬is_ulam) alloc_quit("is_ulam", uncompressed_bytes);
  bytes += uncompressed_bytes ∗ sizeof(uchar);
  u = (14 ∗ maxn − 1)/bits_per_compressed_byte + 1;
  is_um = (uchar ∗) malloc(u ∗ sizeof(uchar));
  if (¬is_um) alloc_quit("is_um", u);
  bytes += u ∗ sizeof(uchar);
  inv_code = (uchar ∗) calloc(1 ≪ bits_per_compressed_byte, sizeof(uchar));
  if (¬inv_code) alloc_quit("inv_code", 1 ≪ bits_per_compressed_byte);
  bytes += (1 ≪ bits_per_compressed_byte) ∗ sizeof(uchar);
  bytes += 256 ∗ sizeof(uint);      /∗ for the preallocated code table ∗/
```

See also sections 22, 26, and 46.

This code is used in section 4.

17. By definition, we know that $U_1 = 1$ and $U_2 = 2$. This gets us started.

⟨ Initialize the data structures 17 ⟩ ≡

```
  ooo, is_ulam[0] = 0, is_ulam[1] = is_ulam[2] = 1;
  cur_slot = uncompressed_bytes, cur_sl = cur_slot/bits_per_compressed_byte;
```

See also sections 27, 31, 36, and 47.

This code is used in section 4.

18. Here in detail is how we test the ulamness of a given x. (We assume implicitly that x is less than the current number u, and that u is at most *cur_slot* + *bits_per_compressed_byte*.)

⟨ Subroutines 10 ⟩ +≡

```
  int ulamq(ullng x) {      /∗ returns nonzero if x is an Ulam number ∗/
    register int c, r, t;
    register ullng q;

    if (x ≥ cur_slot) return (cur_code & (1 ≪ (x − cur_slot)));
    if (x < uncompressed_bytes) return is_ulam[x];
    q = x/bits_per_compressed_byte, r = x mod bits_per_compressed_byte;
    o, c = is_um[q];
    o, t = code[c];
    return t & (1 ≪ r);
  }
```

19. When we've decided the ulamness of u, we enter it into the tables in the following way.

⟨ Record *ulamness* in the *is_ulam* or *is_um* table 19 ⟩ ≡

```
if (u < cur_slot) o, is_ulam[u] = ulamness;
else if (u ≡ cur_slot + bits_per_compressed_byte) ⟨ Store cur_code and get ready for another 20 ⟩
else if (ulamness) cur_code += 1 ≪ u − cur_slot;
```

This code is used in section 32.

20. We always have $code[0] = 0$.

⟨ Store *cur_code* and get ready for another 20 ⟩ ≡

```
{
  o, t = inv_code[cur_code];
  if (¬t) {
    if (cur_code) ⟨ Define a new code t 21 ⟩
    else if (¬code_example[0]) code_example[0] = cur_slot;
  }
  o, is_um[cur_sl] = t;
  code_use[t]++;     /* no mem charged for diagnostic stats */
  cur_sl++, cur_slot += bits_per_compressed_byte;
  cur_code = ulamness;
}
```

This code is used in section 19.

21. ⟨ Define a new code t 21 ⟩ ≡

```
{
  if (code_ptr ≡ 256) {
    fprintf(stderr, "Oops,␣we␣need␣more␣than␣256␣codes!␣You␣must␣decrease␣b.\n");
    goto finish_up;
  }
  o, t = inv_code[cur_code] = code_ptr;
  code_example[code_ptr] = cur_slot;     /* no mem charged */
  o, code[code_ptr++] = cur_code;
}
```

This code is used in section 20.

22. Remembering key Ulam numbers. Continuing at the low level, let's implement the other data structures that record important facts about the Ulam numbers we've seen.

First there's the *window* table, which is easy: It is simply a cyclic buffer for the most recent *window_size* Ulam numbers discovered.

⟨ Allocate the arrays 16 ⟩ +≡
```
window = (ullng *) malloc(window_size * sizeof(ullng));
if (¬window) alloc_quit("window", window_size);
bytes += window_size * sizeof(ullng);
```

23. We'll maintain the value $nw = n \bmod window\_size$.

⟨ Place u into the *window* 23 ⟩ ≡
```
o, window[nw] = u;
```
This code is used in section 37.

24. The other structures, which remember the outliers and near-outliers that have been discovered so far, are more interesting. We need to process those numbers in order of their residues.

Gibbs introduced a special data structure for them, using an index into a doubly linked list. A similar but simpler structure is implemented here, with *two* indexes into two *singly* linked lists.

The number of outliers and near-outliers is, fortunately, small enough that we needn't be too fussy about saving memory space when we store them. Each node of a search list has three fields: Two for the number itself and its residue; one for a link to the successor node.

⟨ Type definitions 9 ⟩ +≡
```
typedef struct {
  ullng u;    /* an Ulam number */
  int r;    /* its residue */
  int next;    /* pointer to the next node in order of r */
} node;
```

25. There are two search lists: One for the outliers and near-outliers with small residues, and one for the outliers and near-outliers with large residues. In the latter we store the complementary residue $\bar{r} = p - r$ instead of r itself as the search key, because we'll be traversing each list in order of increasing keys.

Nodes with the same r key are ordered by their u values.

All nodes of these lists appear in the *nmem* array, with their list heads *lo_out* and *hi_out* in positions 0 and 1.

```
#define bar(r) (lamp − (r))
#define lo_out 0
#define hi_out 1
```

⟨ Global variables 6 ⟩ +≡
```
ullng *window;    /* a cyclic buffer that remembers recent Ulam numbers */
int nw;    /* n mod window_size */
node *nmem;    /* the nodes of binary search trees */
int node_ptr = 2;    /* this many nodes are in use */
uint *inx[2];    /* indexes to the lists */
uint avail;    /* head of the stack of available nodes */
stat ins_stats[4];    /* statistics for insertion into the four trees */
```

26. ⟨ Allocate the arrays 16 ⟩ +≡

```
nmem = (node *) malloc((2 + outliers) * sizeof(node));
if (¬nmem) alloc_quit("nmem", outliers);
bytes += (2 + outliers) * sizeof(node);
inx[0] = (uint *) malloc((isize/2 + 1) * sizeof(uint));
if (¬inx[0]) alloc_quit("inx[0]", outliers);
inx[1] = (uint *) malloc((isize/2 + 1) * sizeof(uint));
if (¬inx[1]) alloc_quit("inx[1]", outliers);
bytes += (isize + 2) * sizeof(uint);
```

27. Lists are terminated either by the *null* link 0 or by the *danger* link 1 (which will be discussed below). Initially the lists are empty, and all index entries point to the list head, whose r field is 0.

```
#define null 0      /* end of list */
#define danger 1      /* end of list that has been cut off */
```

⟨ Initialize the data structures 17 ⟩ +≡

```
oo, nmem[lo_out].next = null, nmem[lo_out].r = 0;
oo, nmem[hi_out].next = null, nmem[hi_out].r = 0;
for (i = 0; i ≤ isize/2; i++) oo, inx[0][i] = lo_out, inx[1][i] = hi_out;
avail = 0;
```

28. Here's now we insert new nodes into such a list. The key invariant is that, if key r causes us to start at index entry j, then every index $j' > j$ will be examined only for keys that are strictly greater than r. Therefore it is legal for them to point to the newly inserted node.

This subroutine is called only when u is larger than any of the u fields already in the list.

```
#define insert(head, u, r)
        if (¬ins(head, u, r)) {
          fprintf(stderr, "Oh␣oh,␣there's␣outlier␣overflow␣(size="O"d)!\n", outliers);
          goto finish_up;
        }
```

⟨ Subroutines 10 ⟩ +≡

```
int ins(int head, ullng u, register int r) {
  register int j, x, y, z, count;
  if (avail) o, z = avail, avail = nmem[avail].next;      /* reuse a recycled node */
  else if (node_ptr < 2 + outliers) z = node_ptr++;
  else return 0;      /* there's no more room */
  oo, nmem[z].u = u, nmem[z].r = r;
  if (vbose & show_outlier_details)
    fprintf(stderr, "␣(remembering␣"O"soutlier␣"O"lld,␣"O"s="O"d)\n",
        r > lamp/3 ? "near-" : "", u, head ≡ hi_out ? "rbar" : "r", r);
  j = ((ullng) r * isize)/lamp;
  o, x = inx[head][j];
  for (o, y = nmem[x].next, count = 1; y > danger ∧ (o, nmem[y].r ≤ r); o, x = y, y = nmem[x].next)
    count++;
  oo, nmem[x].next = z, nmem[z].next = y;
  for (j++; j ≤ isize/2; j++, count++) {
    if (oo, nmem[inx[head][j]].r > r) break;
    o, inx[head][j] = z;
  }
  record_stat(&ins_stats[head], count, u);
  return 1;
}
```

29. We will also sometimes discard a near-outlier, if it becomes more "in" than a discarded inlier. This is where *danger* creeps in to the data.

Again, this subroutine is called only when u is larger than any of the u fields already in the list.

We will never insert items with residue $\geq r$ again, so there's no need to update the index.

```
⟨ Subroutines 10 ⟩ +≡
  void delete (int head, ullng u, register int r) {
    register int j, x, y, count;
    ullng uu;
    j = ((ullng) r * isize)/lamp;
    o, x = inx[head][j];
    for (o, y = nmem[x].next, count = 1; y > danger ∧ (o, nmem[y].r ≤ r); o, x = y, y = nmem[x].next)
      count++;
    o, nmem[x].next = danger;      /* cut off all further elements */
    if (y > danger) {
      for (x = y; o, nmem[y].next > danger; count++) {
        if (vbose & show_outlier_details) {
          r = nmem[y].r, uu = nmem[y].u;      /* no mem charged for diagnostics */
          fprintf(stderr, "␣(forgetting␣"O"soutlier␣"O"lld,␣"O"s="O"d)\n",
                r > lamp/3 ? "near-" : "", uu, head ≡ hi_out ? "rbar" : "r", r);
        }
        y = nmem[y].next;
      }
      o, nmem[y].next = avail, avail = x;
    }
    record_stat(&ins_stats[head], count, u);
  }
```

30. That index and link mechanism is somewhat tricky, so I'd better have a subroutine to check that it isn't messed up.

```
#define flag #80000000   /* flag temporarily placed into the next fields */
#define panic(m)
        {
          fprintf(stderr, "Oops, %s! (h=%d, r=%d, j=%d, x=%d)\n", m, h, r, j, x);
          return;
        }
⟨Subroutines 10⟩ +≡
  void sanity(void)
  {
    register int h, j, nextj, x, y, r, lastr;
    ullng u, lastu;
    for (h = lo_out; h ≤ hi_out; h++) {
      lastr = 0, lastu = 0, j = 1;
      for (x = h; ; x = y) {
        r = nmem[x].r, u = nmem[x].u, y = nmem[x].next;
        if (r < lastr ∨ (r ≡ lastr ∧ u < lastu)) panic("Out of order");
        nextj = ((ullng) r * isize)/lamp;
        for ( ; j ≤ nextj; j++)
          if (¬(nmem[inx[h][j]].next & flag)) panic("Index bad");
        nmem[x].next = y + flag;
        if (y ≤ danger) break;
        lastr = r, lastu = u;
      }
      for (x = h; ; x = y) {
        y = nmem[x].next − flag;
        nmem[x].next = y;
        if (y ≤ danger) break;
      }
    }
  }
```

31. Our assumption that $\lfloor (p-1)/q \rfloor = 2$ ensures that $U_1 = 1$ is a low near-outlier and that $U_2 = 2$ is a high outlier.

Fine point: Since 1 and 2 cannot be expressed as a sum of distinct Ulam numbers, they are Ulam misses as well as Ulam numbers.

```
⟨Initialize the data structures 17⟩ +≡
  oo, window[1] = 1, window[2] = 2;
  n = nw = misses = 2;
  insert(lo_out, 1, lamq);
  insert(hi_out, 2, bar(2 * lamq));
  if (spacing ≡ 1) printf("U1=1\n");
  if (spacing ≡ 1 ∨ spacing ≡ 2) printf("U2=2\n");
```

32. The brute-force tests. Now we're ready to attack the main problem, which is to decide if the current number u is an Ulam number, an Ulam miss, or neither. Gibbs's strategy, as stated above, is to do this in two different ways, depending on u's residue r. Half of the time, when $r \le lamp/4$ or $lamp - r \le lamp/4$, a brute-force search using the previously windowed results will suffice.

⟨ Decide whether u is an Ulam number or an Ulam miss or neither, and update the data structures accordingly 32 ⟩ ≡

```
{
  ⟨ Compute u's residue, r 33 ⟩;
  hits = 0;      /* this is the number of solutions we've found to u = u' + u'' */
  if (r ≤ lamp ≫ 2 ∨ bar(r) ≤ lamp ≫ 2) ⟨ Decide the question via brute force 34 ⟩
  else ⟨ Decide the question via outlier testing 48 ⟩;
ulam_miss: misses++;
  miss_bin[n/alpha][r/beta]++;
not_ulam: ulamness = 0;
  goto finish;
ulam_yes: yes_bin[n/alpha][r/beta]++;
  ⟨ Record u as the next Ulam number 37 ⟩;
  ulamness = 1;
finish: ⟨ Record ulamness in the is_ulam or is_um table 19 ⟩;
}
```

This code is used in section 4.

33. The residue must be computed in two steps, because $lamq * u$ will exceed 64 bits when u is sufficiently large.

⟨ Compute u's residue, r 33 ⟩ ≡

```
  r = u mod lamp;
  r = (lamq * (ullng) r) mod lamp;
```

This code is used in section 32.

34. The brute-force search uses the simple idea that we can have $u = u' + u''$ with $u' > u''$ only if $u' > u/2$. So we look at the previously computed numbers $u' = U_n, U_{n-1}, \ldots$, until we've either found two cases with $u - u'$ an Ulam number, or u' is too small, or we run out of suitable numbers in the window.

⟨ Decide the question via brute force 34 ⟩ ≡

```
{
  x = nw;
  for (o, up = window[x], count = 1; up > (u >> 1); o, up = window[x]) {
    if (ulamq(u - up)) {     /* we found a new solution to u = u' + u'' */
      if (hits) {     /* u not uniquely represented */
        record_stat(&window_stats, count, u);
        goto not_ulam;
      }
      hits = 1;
    }
    if (++count > window_size) {
      fprintf(stderr, "Oh␣oh,␣there's␣window␣overflow␣(size="O"lld)!\n", window_size);
      goto finish_up;
    }
    if (x) x--; else x = window_size - 1;
  }
  record_stat(&window_stats, count, u);
  if (vbose & show_brute_winners) fprintf(stderr,
      "␣(in␣brute-force␣phase,␣"O"lld␣is␣an␣Ulam␣"O"s)\n", u, hits ? "number" : "miss");
  if (hits) goto ulam_yes;
}
```

This code is used in section 32.

35. Histograms for the Ulam numbers and Ulam misses are kept in the arrays *yes_bin* and *miss_bin*, which are of size 16 × 128. (The first index determines the color in the METAPOST illustrations; the second determines the percentage point in the range of r.)

```
#define bincolors 16
#define binsize 128
```

⟨ Global variables 6 ⟩ +≡

```
  stat window_stats;     /* a record of window loop times */
  ullng yes_bin[bincolors][binsize], miss_bin[bincolors][binsize];
  ullng alpha;     /* scale factor for the first index */
  int beta;     /* scale factor the second index */
```

36. ⟨ Initialize the data structures 17 ⟩ +≡

```
  alpha = ((maxn - 1)/bincolors) + 1, beta = ((lamp - 1)/binsize) + 1;
  yes_bin[0/alpha][lamq/beta] = 1, miss_bin[0/alpha][lamq/beta] = 1;
  yes_bin[1/alpha][(2 * lamq)/beta] = 1, miss_bin[1/alpha][(2 * lamq)/beta] = 1;
```

37. Absorbing a new Ulam number. When we've discovered that $U_{n+1} = u$, we celebrate in various ways.

First we increase n and put u into the window.

⟨ Record u as the next Ulam number 37 ⟩ ≡
  n++, nw++;
  if ($nw \equiv window\_size$) $nw = 0$;
  ⟨ Place u into the *window* 23 ⟩;

See also sections 38, 43, and 44.

This code is used in section 32.

38. Next we must decide whether u is an outlier or nearly so.

⟨ Record u as the next Ulam number 37 ⟩ +≡
  if ($r \le lamp/3$) ⟨ Record u as a low outlier 39 ⟩
  else if ($r \le lamp/2$) ⟨ If u is a low near-outlier, record it 41 ⟩
  else if ($bar(r) \le lamp/3$) ⟨ Record u as a high outlier 40 ⟩
  else ⟨ If u is a high near-outlier, record it 42 ⟩;

39. ⟨ Record u as a low outlier 39 ⟩ ≡

```
{
  if (r ≤ lowest_outlier) {
    lowest_outlier = r;
    if (vbose & show_record_outliers)
      fprintf(stderr, "␣(record␣low␣outlier␣r="O"d,␣u="O"lld)\n", r, u);
  }
  insert(lo_out, u, r);
}
```

This code is used in section 38.

40. ⟨ Record u as a high outlier 40 ⟩ ≡

```
{
  if (r ≥ highest_outlier) {
    highest_outlier = r;
    if (vbose & show_record_outliers)
      fprintf(stderr, "␣(record␣high␣outlier␣r="O"d,␣u="O"lld\n", r, u);
  }
  insert(hi_out, u, bar(r));
}
```

This code is used in section 38.

41. Gibbs's heuristic "inness" score, $(\rho - \frac{1}{3})\sqrt{u}$ when $\rho \le \frac{1}{2}$, must be T or less if u is to be remembered as a low near-outlier. We know that $r \ge (p+1)/3$ at this point; hence $T/(\rho - 1/3) = 3Tp/(3r - p) \le 3Tp$.

When we do *not* store u, we must ensure that a mistake hasn't been made. So we will flag an error if any future search for a near-outlying "anchor point" would have encountered a number whose residue is greater than r, or equal to r with an associated value greater than u. (Because in such a case, the algorithm should have really encountered the number we're dropping.)

Think about this carefully, because it's the most subtle point of the program!

We prevent such errors by cutting off the search lists, and recognizing *danger* when we encounter it. We also retain *lo_r_bound*, remembering where cutoffs have previously occurred.

⟨ If u is a low near-outlier, record it 41 ⟩ ≡

```
{
  register double g = lampthresh/((ullng)(3 * r - lamp));
  if (u ≥ g * g) {    /* not near, so we'll drop it */
    if (vbose & show_omitted_inliers)
      fprintf(stderr, "␣(omitting␣r=%d,␣u=%lld,␣g=%g)\n", r, u, g * g/u);
    if (r < lo_r_bound) {
      lo_r_bound = r;
      if (vbose & show_record_cutoffs)
        fprintf(stderr, "␣(record␣low␣cutoff␣r=%d,␣u=%lld,␣g=%g)\n", r, u, g * g/u);
      delete(lo_out, u, r);
    }
  } else if (r < lo_r_bound) insert(lo_out, u, r);
}
```

This code is used in section 38.

42. ⟨ If u is a high near-outlier, record it 42 ⟩ ≡

```
{
  register double g = lampthresh/((ullng)(3 * bar(r) - lamp));
  if (u ≥ g * g) {    /* not near, so we'll drop it */
    if (vbose & show_omitted_inliers)
      fprintf(stderr, "␣(omitting␣rbar=%d,␣u=%lld,␣g=%g)\n", bar(r), u, g * g/u);
    if (bar(r) < hi_r_bound) {
      hi_r_bound = bar(r);
      if (vbose & show_record_cutoffs) fprintf(stderr,
            "␣(record␣high␣cutoff␣rbar=%d,␣u=%lld,␣g=%g)\n", bar(r), u, g * g/u);
      delete(hi_out, u, bar(r));
    }
  } else if (bar(r) < hi_r_bound) insert(hi_out, u, bar(r));
}
```

This code is used in section 38.

43. Next we look at the gap between u and the previous Ulam number, *prevu*.

⟨ Record u as the next Ulam number 37 ⟩ +≡

```
j = u − prevu;
if (j > maxgap) gapcount[maxgap + 1]++;
else gapcount[j]++;
if (j ≥ biggestgap) {
  biggestgap = j;
  if (vbose & show_record_gaps)
    fprintf(stderr, " (gap "O"d = U"O"lld-U"O"lld, U"O"lld="O"lld)\n", j, n, n − 1, n − 1, prevu);
}
prevu = u;
```

44. Finally, we report u itself, if n is a multiple of *spacing*. Other statistics are also printed to *stderr*, if requested.

⟨ Record u as the next Ulam number 37 ⟩ +≡

```
if (spacing ∧ (n mod spacing ≡ 0)) {
  register clock_t t = clock();
  printf("U"O"lld="O"lld\n", n, u);
  if (vbose & show_usage_stats) fprintf(stderr, " ("O"lld misses, "O"lld mems, "O".2f sec)\n",
        misses − prevmisses, mems − prevmems, (double)(t − prevclock)/(double) CLOCKS_PER_SEC);
  prevmisses = misses, prevmems = mems, prevclock = t;
}
```

45. We'd better declare the variables that we've been using.

⟨ Global variables 6 ⟩ +≡

```
double lampthresh;   /* lamp * thresh */
int lowest_outlier, highest_outlier;   /* extreme outliers */
ullng prevu;   /* the Ulam number most recently found */
ullng *gapcount;   /* how often each gap has occurred */
int rbound, rbarbound;   /* search limits on the residue */
ullng ubound;   /* search limits on the value, when residue is max */
int anchorx;   /* the node corresponding to the unique u' with u = u' + u'' */
int lo_r_bound, hi_r_bound;   /* residues at which we've cut data off */
ullng prevmisses;   /* the number of misses most recently reported */
ullng prevmems;   /* the number of mems most recently reported */
clock_t prevclock;   /* the number of microseconds most recently reported */
stat lo_out_stats, hi_out_stats;
int ulamness;   /* is u an Ulam number? */
```

46. ⟨ Allocate the arrays 16 ⟩ +≡

```
gapcount = (ullng *) malloc((maxgap + 2) * sizeof(ullng));
if (¬gapcount) alloc_quit("gapcount", maxgap);
bytes += (maxgap + 2) * sizeof(ullng);
```

47. And we'd better initialize them too.

⟨ Initialize the data structures 17 ⟩ +≡

```
lampthresh = lamp * thresh;
lowest_outlier = lo_r_bound = hi_r_bound = lamp;
highest_outlier = 2 * lamq;
gapcount[1] = 1;
prevu = 2;
```

48. The residue-based tests. OK, we're ready to tackle the main loop of the calculation. I should really say "main loops" (plural), because we use two search lists in this process.

If a unique solution to $u = u' + u''$ is found, *anchorx* will be the node corresponding to u'.

```
⟨ Decide the question via outlier testing 48 ⟩ ≡
  ⟨ Try to decide by anchoring in lo_out 49 ⟩;
  ⟨ Try to decide by anchoring in hi_out 50 ⟩;
  if (hits) {
    if (nmem[anchorx].r > lamp/3 ∧ (vbose & show_inlier_anchors))
      fprintf(stderr, "␣(inlier␣anchor␣U"O"lld="O"lld+"O"lld)\n", n, nmem[anchorx].u,
          u − nmem[anchorx].u);
    goto ulam_yes;
  }
```

This code is used in section 32.

49. If $u = u' + u''$ and $r = r' + r''$, we can assume that $r' \le r''$, hence $r' \le r/2$. Furthermore if $r' = r''$ we can assume that $u' < u''$, hence $u' < u/2$. These facts limit the search, and keep us from finding the same solution twice.

```
⟨ Try to decide by anchoring in lo_out 49 ⟩ ≡
  rbound = r ≫ 1, ubound = (u − 1) ≫ 1;
  for (o, x = nmem[lo_out].next, count = 1; ; o, x = nmem[x].next, count++) {
    if (x ≤ danger) break;
    oo, rp = nmem[x].r, up = nmem[x].u;
    if (rp ≥ rbound) {
      if (rp > rbound ∨ (rp + rp ≡ r ∧ up > ubound)) break;
    }
    o, up = nmem[x].u;
    if (ulamq(u − up)) {      /* we found a new solution to u = u' + u'' */
      if (hits) {
        record_stat(&lo_out_stats, count, u);
        goto not_ulam;
      }
      hits = 1, anchorx = x;
    }
  }
  record_stat(&lo_out_stats, count, u);
  if (x ≡ danger) {
    fprintf(stderr, "Sorry,␣the␣T␣threshold␣is␣too␣low!\n");
    fprintf(stderr, "␣(r="O"d,u="O"lld,lo_r_bound="O"d)\n", r, u, lo_r_bound);
    goto finish_up;
  }
```

This code is used in section 48.

50. Similar observations apply when we're solving $u = u' + u''$, $\bar{r} = \bar{r}' + \bar{r}''$.

⟨ Try to decide by anchoring in *hi_out* 50 ⟩ ≡

```
rbarbound = bar(r) ≫ 1;
for (o, x = nmem[hi_out].next, count = 1; ; o, x = nmem[x].next, count++) {
  if (x ≤ danger) break;
  oo, rp = nmem[x].r, up = nmem[x].u;
  if (rp ≥ rbarbound) {
    if (rp > rbarbound ∨ (rp + rp ≡ bar(r) ∧ up > ubound)) break;
  }
  if (ulamq(u − up)) {     /* we found a new solution to u = u' + u'' */
    if (hits) {
      record_stat(&hi_out_stats, count, u);
      goto not_ulam;
    }
    hits = 1, anchorx = x;
  }
}
record_stat(&hi_out_stats, count, u);
if (x ≡ danger) {
  fprintf(stderr, "Sorry,␣the␣T␣threshold␣is␣too␣low!\n");
  fprintf(stderr, "␣(rbar="O"d,u="O"lld,hi_r_bound="O"d)\n", bar(r), u, hi_r_bound);
  goto finish_up;
}
```

This code is used in section 48.

51. Finishing up. When we're done, we publish the requested subsets of everything that we've learned.

⟨ Print farewell messages 51 ⟩ ≡

```
if (n ≡ maxn ∧ ¬(spacing ∧ (n mod spacing ≡ 0))) printf("U"O"lld="O"lld\n", n, u − 1);
      /* that statement prints the final answer, if not already printed */
if (n < maxn) fprintf(stderr, "I␣found␣"O"lld␣Ulam␣numbers␣and", n);
else fprintf(stderr, "I␣found");
fprintf(stderr, "␣"O"lld␣Ulam␣misses␣<␣"O"lld.\n", misses, u);
if (vbose & show_gap_stats) ⟨ Print the gap statistics 52 ⟩;
if (vbose & show_histograms) ⟨ Print the histograms 53 ⟩;
if (vbose & show_compression_stats) ⟨ Print the compression statistics 54 ⟩;
if (vbose & show_usage_stats) ⟨ Print statistics re time and space 55 ⟩;
```

This code is used in section 4.

52. ⟨ Print the gap statistics 52 ⟩ ≡

```
{
  fprintf(stderr, "******␣Gap␣statistics␣thru␣U"O"lld␣******\n", n);
  for (j = 1; j ≤ maxgap; j++)
    if (gapcount[j]) fprintf(stderr, ""O"5d:"O"14lld\n", j, gapcount[j]);
  if (gapcount[maxgap + 1]) fprintf(stderr, ">"O"4d:"O"14lld\n", maxgap, gapcount[maxgap + 1]);
}
```

This code is used in section 51.

53. ⟨ Print the histograms 53 ⟩ ≡

```
{
  fprintf(stderr, "******␣Histograms␣thru␣U"O"lld␣******\n", n);
  fprintf(stderr, "␣Hits:\n");
  for (j = 0; j < binsize; j++) {
    for (i = 0, u = 0; i < bincolors; i++) u += yes_bin[i][j];
    if (u) fprintf(stderr, ""O"4d/"O"d:"O"14lld\n", j, binsize, u);
  }
  fprintf(stderr, "␣Misses:\n");
  for (j = 0; j < binsize; j++) {
    for (i = 0, u = 0; i < bincolors; i++) u += miss_bin[i][j];
    if (u) fprintf(stderr, ""O"4d/"O"d:"O"14lld\n", j, binsize, u);
  }
}
```

This code is used in section 51.

54. ⟨ Print the compression statistics 54 ⟩ ≡

```
{
  fprintf(stderr, "******␣Compression␣summary:␣******\n");
  for (j = (code_use[0] ? 0 : 1); j < code_ptr; j++) {
    fprintf(stderr, "␣"O"02x␣", j);
    for (k = 1 ≪ (bits_per_compressed_byte − 1); k; k ≫= 1) fprintf(stderr, ""O"d", code[j] & k ? 1 : 0);
    fprintf(stderr, ""O"14lld"O"14lld\n", code_use[j], code_example[j]);
  }
}
```

This code is used in section 51.

55. **#define** *dump_stats*(*st*)
    fprintf(*stderr*, "n␣"*O*"lld,␣mean␣"*O*"g,␣max␣"*O*"d␣("*O*"lld)\n", *st*.*n*, *st*.*mean*, *st*.*max*, *st*.*ex*);

⟨ Print statistics re time and space 55 ⟩ ≡

```
{
  fprintf(stderr, "\nBrute-force␣loop␣stats:␣");
  dump_stats(window_stats);
  fprintf(stderr, "Low-outlier␣insertion␣stats:␣");
  dump_stats(ins_stats[lo_out]);
  fprintf(stderr, "Low-outlier␣loop␣stats:␣");
  dump_stats(lo_out_stats);
  fprintf(stderr, "High-outlier␣insertion␣stats:␣");
  dump_stats(ins_stats[hi_out]);
  fprintf(stderr, "High-outlier␣loop␣stats:␣");
  dump_stats(hi_out_stats);
  fprintf(stderr, "The␣outlier␣lists␣used␣"O"d␣cells.\n", node_ptr − 2);
  fprintf(stderr, "Altogether␣"O"lld␣bytes,␣"O"lld␣mems,␣"O".2f␣sec.\n", bytes, mems, (double)
      clock()/(double) CLOCKS_PER_SEC);
}
```

This code is used in section 51.

56. The METAPOST output. Pretty pictures, comin' right up.

⟨ Output the METAPOST file 56 ⟩ ≡

```
{
  fprintf(mp_file, ""O""O"␣created␣by␣gibbs-ulam␣"O"lld\n", maxn);
  ⟨ Output the boilerplate 58 ⟩;
  factor = (double)(binsize * binsize)/((double) 9 * maxn);
  fprintf(mp_file, "\nbeginfig(1)␣init;␣"O""O"␣distribution␣of␣Ulam␣numbers\n");
  for (j = 0; j < binsize; j++) acc[j] = 0, prev[j] = 0;
  for (i = bincolors − 1; i ≥ 0; i−−) {
    fprintf(mp_file, "doit("O"d)\n␣␣", i);
    for (j = 0; j < binsize; j++) {
      acc[j] += yes_bin[i][j];
      t = (int)(factor * acc[j] + 0.5);
      fprintf(mp_file, ""O"d"O"s", t − prev[j],
          j + 1 ≡ binsize ? ";\n" : (j & #f) ≡ #f ? ",\n␣␣" : ",");
      prev[j] = t;
    }
  }
  fprintf(mp_file, "endfig;\n\n");
  fprintf(mp_file, "beginfig(0)␣init;␣"O""O"␣distribution␣of␣Ulam␣misses\n");
  for (j = 0; j < binsize; j++) acc[j] = 0, prev[j] = 0;
  for (i = bincolors − 1; i ≥ 0; i−−) {
    fprintf(mp_file, "doit("O"d)\n␣␣", i);
    for (j = 0; j < binsize; j++) {
      acc[j] += miss_bin[i][j];
      t = (int)(factor * acc[j] + 0.5);
      fprintf(mp_file, ""O"d"O"s", t − prev[j],
          j + 1 ≡ binsize ? ";\n" : (j & #f) ≡ #f ? ",\n␣␣" : ",");
      prev[j] = t;
    }
  }
  fprintf(mp_file, "endfig;\n\nbye.\n");
  close(mp_file);
  fprintf(stderr, "METAPOST␣code␣written␣to␣file␣"O"s.\n", mp_name);
}
```

This code is used in section 4.

57. ⟨ Global variables 6 ⟩ +≡

```
ullng acc[binsize];  /* accumulated histogram data */
int prev[binsize];  /* previously output and rounded histogram data */
double factor;  /* scale factor for histogram data in the METAPOST output */
```

58. ⟨ Output the boilerplate 58 ⟩ ≡

```
fprintf (mp_file, "newinternal␣n;␣numeric␣a[];\n\n");
fprintf (mp_file, "def␣init␣=\n␣␣draw␣(1,0)--("O"d,0);\n", binsize);
fprintf (mp_file, "␣␣for␣j=1␣upto␣"O"d:␣a[j]:=0;␣endfor\n", binsize);
fprintf (mp_file, "␣␣pickup␣pencircle;\nenddef;\n\n");
fprintf (mp_file, "def␣doit(text␣j)␣text␣l␣=\n");
fprintf (mp_file, "␣␣drawoptions(withcolor␣j/"O"d[green,red]);\n", bincolors);
fprintf (mp_file, "␣␣n:=1;\n");
fprintf (mp_file, "␣␣for␣t=l:\n");
fprintf (mp_file, "␣␣␣if␣t>0:␣draw␣(n,a[n])--(n,a[n]+t);␣a[n]:=a[n]+t;␣fi\n");
fprintf (mp_file, "␣␣␣n:=n+1;\n");
fprintf (mp_file, "␣␣endfor\nenddef;\n");
```

This code is used in section 56.

59. Index.

ULAM-GIBBS