Counting Magic and Multimagic Series

Lee Morgenstern

October, 2013

Table of Contents

Introduction

Both magic and multimagic series can be efficiently counted
using similar recursive techniques.

Magic series up to order 30 can be counted in less than one second.
Bimagic series up to order 14 can be counted in less than one second.
Bimagic series up to order 28 can be counted in just a few hours.

The storage requirement is really the main issue about the feasibility
of counting higher orders of bimagic series.
Order 21 requires 880 Mb of list storage and 330 Mb of table storage.
Order 22 requires 1350 Mb of list storage and 435 Mb of table storage.
Each higher order takes about 8/5 times more list storage than the previous order.
Order 28 needs a computer with 24 Gb of memory.

Trimagic, tetramagic series, and so on, take longer to count,
increase per order faster than bimagic series and require considerably more storage.

Definitions

N = magic series order
N^2 = number of elements
e is an element ranging from 1 ... N^2
V(e) = value of element e
S1 = magic sum
S2 = magic sum of squares
S3 = magic sum of cubes
etc.

Recursive Functions
Magic Series

Define
C(r,s,e) = count of magic series with sum s
using some subset of r elements from e ... N^2.
Then
C(N,S1,1) = total count of magic series.

Note that
C(r,s,e+1) = amount of C(r,s,e) that doesn't use element e,
C(r-1, s-V(e), e+1) = amount of C(r,s,e) that uses element e,
thus
C(r,s,e) = C(r,s,e+1) + C(r-1, s-V(e), e+1).

This gives us a recursive function for computing the counts.
We proceed in stages from e = N^2 down to e = 1.
All the counts for stage e can be computed from counts of stage e+1.

Multimagic Series

Define
L(r,s,e) = list of records, each record containing
{ count, sum2, sum3, etc. },
where the count is the number of multimagic series
having sum s, sum of squares sum2, sum of cubes sum3, etc.,
using some subset of r elements from e ... N^2.

Then
L(N,S1,1) should contain the record { count, S2, S3, etc. }
having the total count of multimagic series.

Note that
L(r,s,e+1) = sublist of L(r,s,e) that doesn't use element e,
L(r-1, s-V(e), e+1) = sublist of L(r,s,e) that uses element e,
but with smaller sum2 values by e^2,
smaller sum3 values by e^3, etc.,
thus
L(r,s,e) = list-merge of L(r,s,e+1) with a modified L(r-1, s-V(e), e+1)
by adding e^2, e^3, etc. to the sums in each of its records.

This gives us a recursive operation for computing multimagic counts.
We proceed in stages from e = N^2 down to e = 1, the same as with magic series.

Reduced Table Implementation

If we implement the tables C[r][s][e] or L[r][s][e],
the table with e values is computed only from the table with e+1 values.
This means that once the table with e values is computed
we no longer need the table with e+1 values.

Furthermore, if we compute a table in reverse order of its r indexes,
we need only one table updated from other parts of the same table.

We can then define the reduced tables C[r][s] or L[r][s],
with rows r = 1 ... N and columns s = 1 ... S1,
and keep track of the value of e using loop control.

Part I. Counting Magic Series

A Rudimentary C Code Implementation

Here is a rudimentary C code implementation for counting magic series.
Improvements will follow.

```
//================================================
#include <stdlib.h>
#include <stdio.h>
static int **C;   // pointer to allocated table count entries
static int *V;    // pointer to allocated list of elements
static int N, emax, S1, e, v;
static int count_magic_series(void);

//------------------------------------------------
void main(int argc, char **argv)
{
  int r;
  int total;

  N = atoi(argv[1]);  // read order from command line
  emax = N*N;         // compute maximum element
  S1 = N*(emax+1)/2;  // compute magic constant

  V = (int *)malloc((emax+1) * sizeof(int));
  for (e = 1; e <= emax; ++e)
    V[e] = e;

  C = (int **)malloc((N+1) * sizeof(int *));
  for (r = 1; r <= N; ++r)
    C[r] = (int *)malloc((S1+1) * sizeof(int));

  total = count_magic_series();

  for (r = N; r >= 1; --r)
    free(C[r]);
  free(C);
  free(V);

  printf("Total = %d\n", total);
}
```

```
//-----------------------------------------------------
static int count_magic_series(void)
{
  int r, s;

  for (r = 1; r <= N; ++r)           // zero all table counts
    for (s = 1; s <= S1; ++s)
      C[r][s] = 0;

  for (e = emax; e >= 1; --e)        // stages emax down to 1
  { int v = V[e];
    for (r = N; r >= 2; --r)         //   update row N down to row 2
      for (s = v+1; s <= S1; ++s)    //     update counts
        C[r][s] += C[r-1][s-v];

    C[1][v] = 1;                     //   update row 1 separately
  }

  return C[N][S1];                   // return final count
}
//=============================================================================
```

Improvements

Improving a time by a few seconds doesn't seem like it's worth the trouble.
However, all of these improvements for counting magic series also apply to counting multimagic series which takes a lot longer.

Reducing the r loop ranges

During early stages, the higher numbered rows have all zeroes.
It wastes time to update a row using the values from an all-zero row.

At the start of stage emax-1, only row 1 has a non-zero entry,
so only rows 1 and 2 need be updated.
At stage emax-2, row 2 has non-zero entries,
so row 3 can then be updated, but not rows 4 thru N.
In general, at stage e, only the rows r <= emax-e+1
need be updated.

The r loop can also be reduced during the later stages when the
lower numbered rows are no longer needed.

After stage e = 1, only row N will be used to read the total count,
thus rows N-1 and lower need not be updated during stage e = 1.
At stage e = 2, only rows N-1 and N need to be updated.
In general, at stage e, the rows r = N-e and lower
need not be updated.

Using the above two principles, the r loop can be reduced as follows.

```
   rmax = min(N, emax-e+1);
   rmin = max(2, N-e+1);
   for (r = rmax; r >= rmin; --r)
```

Reducing the s loop by limiting the loop end index

The following assumes that V[e] is ordered so that V[e+1] > V[e] for all e.

During stage e = emax-1, row 2 will be updated for the first time using

```
  v = V[emax-1];
  for (s = v+1; s <= S1; ++s)
    C[2][s] += C[1][s-v];
```

Since row 1 will have a single non-zero entry at C[1][V[emax]],
a single non-zero entry in row 2 will occur at C[2][V[emax]+V[emax-1]].
All entries following this one will be zero and remain zero
for subsequent stages.

During stage e = emax-2, row 3 will be updated for the first time using

```
  v = V[emax-2];
  for (s = v+1; s <= S1; ++s)
    C[3][s] += C[2][s-v];
```

Since row 2 will have a single non-zero entry at C[2][V[emax]+V[emax-1]],
a single non-zero entry in row 3 will occur at C[3][V[emax]+V[emax-1]+V[emax-2]],
while entries following this one will be zero for this stage
and all later stages.

Define T[r] = sum of largest r values of V[].

```
  T[0] = 0;
  for (r = 1; r <= N; ++r)
    T[r] = T[r-1] + V[emax-r+1];
```

In general, the first non-zero entry in row r will occur at C[r][T[r]].
All entries following this one will always be zero.

Since the entries in row r-1 beyond C[r-1][T[r-1]] will always be zero,
it is a waste of time to use those entries to update row r.
So we can limit the s range when updating row r to this.

```
  v = V[e];
  smaxA = T[r-1] + v;
  smax = min(smaxA, S1);
  for (s = v+1; s <= smax; ++s)
     C[r][s] += C[r-1][s-v];
```

Reducing the s loop by advancing the loop start index

After stage e+1, row 1 will have non-zero entries starting from C[1][V[e+1]],
so the stage e update of row 2 can be reduced to using just those entries.

```
   v = V[e];
   sminA = V[e+1]+V[e];
   for (s = sminA; s <= smax; ++s)
     C[2][s] += C[1][s-v];
```

After stage e+1, row 2 will have non-zero entries starting from C[2][V[e+2]+V[e+1]],
so the stage e update of row 3 can be reduced as follows.

```
   v = V[e];
   sminA = V[e+2]+V[e+1]+V[e];
   for (s = sminA; s <= smax; ++s)
     C[3][s] += C[2][s-v];
```

Define U[r] = sum of smallest r values of V[].

```
   U[0] = 0;
   for (r = 1; r <= N; ++r)
     U[r] = U[r-1] + V[r];
```

In general, after stage e+1, row r-1 will have non-zero entries starting from
C[r-1][U[r+e-1]-U[e]], where U[n] = sum of smallest n values of V[],
so the stage e update of row r can be reduced as follows.

```
   v = V[e];
   sminA = U[r+e-1] - U[e-1];
   for (s = sminA; s <= smax; ++s)
     C[r][s] += C[r-1][s-v];
```

Reducing the s loop for row N

The entries in row N are not used to update any other row.
After the last stage, we need to read only one entry from row N, C[N][S1].
Thus, we can eliminate the row N update loop and replace it with
one statement updating just that one needed entry.

```
   v = V[e];
   C[N][S1] += C[N-1][S1-v];
```

Reducing the s loop ranges even further

The range of (S1-v) entries from row N-1 that can update that one entry in row N
is from (S1-V[emax]) to (S1-V[1]). So these are the only entries in row N-1 that
need updating from row N-2. But this range can be reduced even further
as the value of e decreases in each successive stage.

After using the entry C[N-1][S1-V[e]] to update row N during stage e,
we no longer need it because the following stages will use C[N-1][S1-V[e-1]]
and then C[N-1][S1-V[e-2]], and so on up to C[N-1][S1-V[1]].
Therefore, during stage e, after row N is updated, we can update row N-1
from row N-2 using this reduced loop range.

```
  v = V[e];
  sminB = S1 - V[e-1];
  smaxB = S1 - V[1];
  for (s = sminB; s <= smaxB; ++s)
    C[N-1][s] += C[N-2][s-v];
```

The last stage for updating row N-1 occurs when e = 2,
thus the largest (s-v) entry from row N-2 that is needed to update entries
in row N-1 is (S1-V[1]-V[2]).

During stage e, the N-2 row will update the N-1 row starting from C[N-2][S1-V[e-1]-V[e]].
The next stage will start from C[N-2][S1-V[e-2]-V[e-1]].
Thus, we can use the following reduced range for updating row N-2.

```
  v = V[e];
  sminB = S1 - V[e-2] - V[e-1];
  smaxB = S1 - V[1] - V[2];
  for (s = sminB; s <= smaxB; ++s)
    C[N-2][s] += C[N-3][s-v];
```

The last stage for updating row N-2 occurs when e = 3,
thus the largest (s-v) entry from row N-3 that is needed to update entries
in row N-2 is (S1-V[1]-V[2]-V[3]).

During stage e, the N-3 row will update the N-2 row starting from C[N-3][S1-V[e-2]-V[e-1]-V[e]].
The next stage will start from C[N-3][S1-V[e-3]-V[e-2]-V[e-1]].
Thus, we can use the following reduced loop for updating row N-3.

```
  v = V[e];
  sminB = S1 - V[e-3] - V[e-2] - V[e-1];
  smaxB = S1 - V[1] - V[2] - V[3];
  for (s = sminB; s <= smaxB; ++s)
    C[N-3][s] += C[N-4][s-v];
```

In general, we can update row r using this range.

```
  k = N-r;
  v = V[e];
  sminB = S1 - (U[e-1] - U[e-k-1]);
  smaxB = S1 - U[k];
```

We then combine these with the other ranges to get the smallest range.

```
  sminA = U[r+e-1] - U[e-1];
  smaxA = T[r-1] + v;

  smin = max(sminA, sminB);
  smax = min(smaxA, smaxB);
  for (s = smin; s <= smax; ++s)
    C[r][s] += C[r][s-v];
```

Here is the improved C code implementation for counting magic series.

```
#include <stdlib.h>
#include <stdio.h>
//---------------------------------------------------------------
static int T[MAX_ORDER+1];
static int U[MAX_ORDER+1];
static int **C;              // pointer to allocated table count entries
static int *V;    // pointer to allocated list of elements
static int N, emax, S1, e, v;
//-----------------------------------------------
static int count_magic_series(void);
static void zero_table_counts(void);
static void update_row_N(void);
static void update_mid_rows(void);
static void update_row_1(void);
//-----------------------------------------------
void main(int argc, char **argv)
{
  int r;
  int total;

  N = atoi(argv[1]);    // read order from command line
  emax = N*N;           // compute maximum element
  S1 = N*(emax+1)/2;    // compute magic constant

  V = (int *)malloc((emax+1) * sizeof(int));
  for (e = 1; e <= emax; ++e)
    V[e] = e;

  C = (int **)malloc((N+1) * sizeof(int *));
  for (r = 1; r <= N; ++r)
    C[r] = (int *)malloc((S1+1) * sizeof(int));

  T[0] = 0;
  for (r = 1; r <= N; ++r)
    T[r] = T[r-1] + V[emax-r+1];

  U[0] = 0;
  for (r = 1; r <= N; ++r)
    U[r] = U[r-1] + V[r];

  total = count_magic_series();

  for (r = N; r >= 1; --r)
    free(C[r]);
  free(C);
  free(V);

  printf("total = %d\n", total);
}
//-----------------------------------------------
static int count_magic_series(void)
{
  zero_table_counts();

  e = emax;
  update_row_1();
```

```
  for (e = emax-1; e >= 1; --e)
  { update_row_N();
    update_mid_rows();
    update_row_1();
  }
  return C[N][S1];  // return final count
}
//-------------------------------------------------
static void zero_table_counts(void)
{
  int r, s;

  for (r = 1; r <= N; ++r)
    for (s = 1; s <= S1; ++s)
      C[r][s] = 0;
}
//-------------------------------------------------
static void update_row_1(void)
{
  if (e >= N)
    C[1][e] = 1;
}
//-------------------------------------------------
static void update_row_N(void)
{
  if (e <= emax-N+1)
    C[N][S1] += C[N-1][S1-e];
}
//-----------------------------------------------
static void update_mid_rows(void)
{
  int r, s, k;
  int smin, sminA, sminB;
  int smax, smaxA, smaxB;
  int rmin, rmax;

  rmax = min(N-1, emax-e+1);
  rmin = max(2, N-e+1);

  k = N - rmax;
  sminA = rmax*e + T[rmax-1];
  smaxA = (rmax-1)*emax - T[rmax-2] + e;
  sminB = S1 - k*e + T[k];
  smaxB = S1 - T[k];

  for (r = rmax; r >= rmin; --r)
  { smin = max(sminA, sminB);
    smax = min(smaxA, smaxB);

    for (s = smin; s <= smax; ++s)
      C[r][s] += C[r-1][s-e];

    sminA -= (e+r-1);
    smaxA -= (emax-r+2);
    ++k;
    sminB -= (e - k);
    smaxB -= k;
  }
}
```

Intermission

Large Count Technique

For large N resulting in super huge counts of magic series
requiring too much storage, we can trade extra computation time
to reduce the storage.

Allocate 32-bit table entries.
Select a large prime number modulus m1 that fits in 31 bits or less.
Compute the table entries modulo m1.
For example,

```
    C[r][s] += C[r-1][s-e];
    if (C[r][s] >= m1)
      C[r][s] -= m1;
```

Save the final value of C[N][S1] (mod m1).

Select a different prime number modulus m2.
Recompute all table entries modulo m2,
Save the final value of C[N][S1] (mod m2).

Keep repeating the above with different prime moduli until the product
of the moduli exceeds the expected number of magic series.

Use the Chinese Remainder Theorem to compute the exact number of
magic series from the multiple modulus residues.
This is the only part that requires multiprecision arithmetic.

Extra speed-up. The Chinese Remainder Theorem requires
only that the moduli be pairwise coprime. Thus one of the
moduli can be a power of 2 such as 2^{32}.
Using (mod 2^{32}) requires no extra code to adjust for a residue.
You just allow the arithmetic to overflow in the 32-bit words.

Part II. Counting Bimagic Series

Replace the C table with the L table.
Instead of a table of counts, it is a table of pointers
to lists of records containing counts and sums of squares.
The update operation is replaced by a list merge operation.

A Rudimentary Implementation

Here is a rudimentary C code implementation for counting bimagic series.
Improvements from the magic series algorithm will be added later.
Bimagic improvements will also be added.

```
//================================================================
typedef struct
{ int cs_count;   // count of bimagic series
  int cs_sum2;    // having this sum of squares
} COUNTSUM;

typedef struct
{ int csl_length;          // number of records in list, followed by
  COUNTSUM csl_record[1];  // variable length list of count/sum2 records
} COUNTSUMLIST;
```

```
static COUNTSUMLIST ***L;  // allocated table of pointers to lists

static int N, emax, S1, S2, e, e2;

static int count_bimagic_series(void);
static COUNTSUMLIST *merge_lists(COUNTSUMLIST *listA, COUNTSUMLIST *listB);

void heap_init(void);           // external support routine
void heap_term(void);           // external support routine
void *heap_allocate(int size);  // external support routine

//-------------------------------------------------
void main(int argc, char **argv)
{
  int r;
  int total;

  N = atoi(argv[1]);               // read order from command line
  emax = N*N;                      // compute maximum element
  S1 = N*(emax+1)/2;               // compute magic constant
  S2 = N*(emax+1)*(2*emax+1) / 6;  // compute bimagic constant

  L = (COUNTSUMLIST ***)malloc((N+1) * sizeof(COUNTSUMLIST **));
  for (r = 1; r <= N; ++r)
    L[r] = (COUNTSUMLIST **)malloc((S1+1) * sizeof(COUNTSUMLIST *));

  heap_init();                      // start memory allocator

  total = count_bimagic_series();

  heap_term();                      // end memory allocator

  for (r = N; r >= 1; --r)
    free(L[r]);
  free(L);

  printf("total = %d\n", total);
}
//-------------------------------------------------
static int count_bimagic_series(void)
{
  int r, s;

  for (r = 1; r <= N; ++r)         // null all table pointers
    for (s = 1; s <= S1; ++s)
      L[r][s] = NULL;

  for (e = emax; e >= 1; --e)      // stages emax down to 1
  {
    e2 = e*e;                      //   compute square of element

    for (r = N; r >= 2; --r)       //   row N down to row 2
      for (s = e+1; s <= S1; ++s)  //     sums e+1 to S1
        L[r][s] = merge_lists(L[r][s], L[r-1][s-e]);

    L[1][e] = (COUNTSUMLIST *)heap_allocate(sizeof(COUNTSUMLIST));
    L[1][e]->csl_length = 1;
    L[1][e]->csl_record[0].cs_count = 1;
    L[1][e]->csl_record[0].cs_sum2 = e2;
  }
```

```
  // search for a solution in the last list
  if (L[N][S1] != NULL)
  { int cnt = L[N][S1]->csl_length;
    COUNTSUM *list = &L[N][S1]->csl_record[0];
    while (cnt--)
    { if (list->cs_sum2 == S2)
        return list->cs_count;    // return final count
      ++list;
    }
  }
  return 0;   // return count of 0 if no records with sum2 == S2 found
}
//-------------------------------------------------------------
static COUNTSUMLIST *merge_lists(COUNTSUMLIST *listA, COUNTSUMLIST *listB)
{
  int cntA, cntB;
  int srcBsum2e2;
  COUNTSUMLIST *dstlist;
  COUNTSUM *srcA, *srcB, *dst;

  if (listB == NULL) return listA;

  cntB = listB->csl_length;
  srcB = &listB->csl_record[0];
  srcBsum2e2 = srcB->cs_sum2 + e2;     // listB records must have e2 added to sum2

  if (listA == NULL)
    cntA = 0;
  else
  { cntA = listA->csl_length;
    srcA = &listA->csl_record[0];
  }

  dstlist = (COUNTSUMLIST *)heap_allocate(sizeof(int) + (cntA + cntB) * sizeof(COUNTSUM));
  dstlist->csl_length = 0;             // init destination list count to 0
  dst = &dstlist->csl_record[0];

  while (cntA > 0 && cntB > 0)         // while both lists non-empty
  {                                    //   copy smaller sum2 record to destination
    if (srcA->cs_sum2 < srcBsum2e2)
    {
      *dst++ = *srcA++;
      ++dstlist->csl_length;
      --cntA;
    }
    else if (srcA->cs_sum2 > srcBsum2e2)
    {
      *dst = *srcB++;
      ++dstlist->csl_length;
      dst->cs_count = srcBsum2e2;
      ++dst;
      if (--cntB)
        srcBsum2e2 = srcB->cs_sum2 + e2;  // maintain sum2 + e2
    }
    else                               // unless both records have equal sum2
    {                                  // in which case ...
      *dst = *srcA++;
      ++dstlist->csl_length;
      dst->cs_count += srcB->cs_count; //   ... accumulate counts in one record
      ++dst;
```

```
      ++srcB;
      --cntA;
      if (--cntB)
        srcBsum2e2 = srcB->cs_sum2 + e2;   // maintain sum2 + e2
    }
  }

  while (cntA--)                 // copy the rest of listA to destination list
  { *dst++ = *srcA++;
    ++dstlist->csl_length;
  }

  while (cntB--)                 // copy the rest of listB to destination list
  { dst->count = srcB->count;
    dst->sum2 = srcB->sum2 + e2;
    ++dst;
    ++srcB;
    ++dstlist->csl_length;
  }

  return dstlist;    // return pointer to destination list
}
//============================================================
```

Improvements

The rudimentary algorithm is very wasteful of memory.
It would be better to store lists into a preallocated circular buffer.
Lists would then be copied from an earlier part of the buffer to a later part
so that the earlier parts can be reused.

The lists for row 1 and row N consist of a single record,
so it might be better to store them into a fixed area of memory.

Modify the merge_lists routine to accept sum2 ranges.
Don't accumulate records that have out-of-range sum2 values.
Besides reducing the storage requirement, this will eliminate the need
for a search of sum2 == S2 at the end of the program.

The merge operation is done when updating the list for a table entry
row and sum and the range of sum values that are updated is sometimes
less than the range that are need for the updated row to do its own update.
This will cause a problem when the circular buffer wraps around
because it will overwrite lists that are needed.
So the update operation needs to also copy lists in a row
that won't be updated but will be needed later.

Note that at the end of the routine count_bimagic_series(),
a search is done to find a record containing sum2 = S2.
Since this is the only use for this list,
it should have at most one record containing sum2 = S2.
This would eliminate the search and also reduce the storage.

When updating the list at L[N][S1], store only records that have sum2 = S2.
An automatic way of doing this is when updating row N-1,
store only sum2 records that will be used to update L[N][S1] with S2.
That is, L[N-1][s] should contain at most one record with sum2 = S2 - $(S1-s)^2$.

When updating a list for rows N-2 and lower, more than one record should be stored.
Low and high sum2 values should be determined and passed in to the merge_lists routine.

```
   static COUNTSUMLIST *merge_lists(COUNTSUMLIST *listA, COUNTSUMLIST *listB,
                                    int min2, int max2);
```

The high and low bounds would be equal when updating row N and N-1.

```
   L[N][S1] = merge_lists(L[N][S1], L[N-1][S1-e], S2, S2);

   d = S1 - s;
   minandmax = S2 - d*d;
   L[N-1][s] = merge_lists(L[N-1][s], L[N-2][s-e], minandmax, minandmax);
```

For updating rows N-2 and lower, min and max values can be determined optimally
by precomputing a 3-dimensional table.

Optimization Technique for Multimagic Series Generation and Counting

Introduction

Backtracking generation and recursive counting of multimagic series
require many checks on whether it is possible to extend a partial series
to a full series. If it is not possible, either backtracking can occur
or the need for storing records can be eliminated.
A strong check can speed up an algorithm and reduce storage requirements.

Described here is a very fast way of precomputing tables of optimal values
that can be used to make the strongest possible checks.

Problem Definition

Here is a definition of the problem using bimagic series as an example.

N = number of elements in a full bimagic series
S1 = target sum of a full bimagic series
S2 = target sum of squares of a full bimagic series

Given a partial bimagic series with r elements having
sum1 = sum of those elements,
sum2 = sum of squares of those elements, and
1 ... e = available range of values to extend the partial bimagic series,

we want to compare sum2 to the maximum and minimum values that are possible
to simultaneously reach S1 and S2 when extending the partial bimagic series
to a full bimagic series. If sum2 is out of range, we can reject the
partial bimagic series as impossible to extend.

Past Solution Attempts

Checks have been used in the past which were not optimal.

Here is a typical check.

Let sum2togo = S2 - sum2 = sum of squares to go.
Let k = N-r = number of elements to go.
Let MaxSumE = sum of squares of largest k elements from 1 ... e.
Let MinSumE = sum of squares of smallest k elements from 1 ... e.

If sum2togo > MaxSumE or sum2togo < MinSumE, then it is impossible
to extend the partial bimagic series into a full bimagic series.

The problem with this check is that it doesn't take into account
reaching S1 and S2 simultaneously. It might be possible to reach S2
by using particular cominbinations of the remaining elements,
but none of those combinations simultaneously reaches S1.

Here is a more sophisticated check using the power mean inequality.
This determines whether it is possible to reach S1 and S2 simultaneously
using the same combination of elements.

Given any set of k positive real numbers, a_1, a_2, ..., a_k,
it is always true that

$$(a_1 + \ldots + a_k)^2 <= k(a_1^2 + \ldots + a_k^2).$$

Let sum1togo = S1 - sum1 = sum to go.
Let sum2togo = S2 - sum2 = sum of squares to go.
Let k = N-r = number of elements to go.

If $(\text{sum1togo})^2$ > (k * sum2togo), then the inequality is violated
and it is impossible to extend the partial bimagic series
into a full bimagic series.

The problem with this check is that it doesn't take into account
the values of the remaining elements. The inequality might be
satisfiable because there exists a set of a_j, but the a_j are
not in the correct range. Another problem is that the a_j are real numbers
and not necessarily distinct, but in our problem, the elements must be
distinct integers.

A much stronger check is possible, $(\text{sum1togo})^2$ + gap >= (k * sum2togo),
where there are minimum and maximum values for the gap based on
distinct integers in the available range. It is known that the
minimum value for the gap based on the use of distinct integers is
$k^2(k^2-1)/12$, but this still does not take into account the values
and range of the available integers.

Problem Solution

Precompute two tables.

```
MinSum2[k][e][s] = minimum value of the sum of squares of elements
                   over all combinations of k distinct elements
                   in the range 1 ... e which sum to s.

MaxSum2[k][e][s] = maximum sum of squares under the same conditions.
```

Let k = N-r
Let s = S1 - sum1
Let sum2togo = S2 - sum2,

If sum2togo < MinSum2[k][e][s] or sum2togo > MaxSum2[k][e][s],
then reject the partial bimagic series as impossible to extend
to a full bimagic series.

Recursive Computation of the Tables

Having a partial bimagic series that needs to be extended by a sum of s,
using k elements from the range 1 ... e, there are two choices.

If a greater sum of squares is possible by not using element e, then
  MaxSum2[k][e][s] = MaxSum2[k][e-1][s].

If a greater sum of squares is possible by using element e, then
  MaxSum2[k][e][s] = e*e + MaxSum2[k-1][e-1][s-e].

Thus we have a recursive operation to compute the table entries.
  MaxSum2[k][e][s] = max (MaxSum2[k][e-1][s], e*e + MaxSum2[k-1][e-1][s-e])

```
  emax = N*N;
  S1 = N*(emax+1)/2;

  k = 1;
  for (s = 1; s <= S1; ++s)
  { s2 = s*s;
    for (e = 1; e < s; ++e) MaxSum2[k][e][s] = -HUGE;
    for (e = s; e <= emax; ++e) MaxSum2[k][e][s] = s2;
  }

  for (k = 2; k <= N; ++k)
  { for (e = 1; e < k; ++e) for (s = 1; s <= S1; ++s) MaxSum2[k][e][s] = -HUGE;
    for (e = k; e <= emax; ++e)
    { e2 = e*e;
      for (s = 1; s <= e; ++s) MaxSum2[k][e][s] = -HUGE;
      for (s = e+1; s <= S1; ++s)
      { a = MaxSum2[k][e-1][s];
        b = MaxSum2[k-1][e-1][s-e];
        MaxSum2[k][e][s] = (b == -HUGE) ? a : max(a,b+e2);
      }
    }
  }
```

The computation of MinSum2[k][e][s] is similar.
Change all instances of max() to min().
Change all instances of -HUGE to +HUGE.

Table Storage Reduction

One of the purposes of the tables is to reduce the storage
requirement for the recursive counting of multimagic series.
But the 3-dimensional tables use a lot of storage themselves.
Fortunately, there is an easy way of reducing the table storage.

Tables for k = 1 and k = N are not used by the main program.
Also, rows within a table such that e < k are not used
by the main program. Some of these, however, are used by the
procedure to generate the table. But the procedure can be
modified so that it doesn't need them. Table k = 1 has a
very simple structure that can be determined as needed when
table k = 2 is produced. Rows where e < k all have
-HUGE in them, thus they can be determined easily.

An Optimized C Code Implementation for Counting Bimagic Series

```
//============================================================================
#include <stdlib.h>
#include <stdio.h>
//----------------------------------------------------------------------------
typedef struct
{ bigint cs_count;   // count of bimagic series
     int cs_sum2;    // having this sum of squares
} COUNTSUM;

typedef struct
{ int csl_length;          // number of records in list, followed by
  COUNTSUM csl_record[1];  // variable length list of records
} COUNTSUMLIST;

typedef struct      // s range for saved values
{ int mm_min;
  int mm_max;
} MINMAX;

static void *StartAllocPtr;  // start of memory allocation for lists
static void *EndAllocPtr;    // end of memory allocation for lists
static void *NextAllocPtr;   // next location to store a list

static int T[MAX_ORDER+1];             // triangular numbers
static int T2[MAX_ORDER*MAX_ORDER+1];  // T2[k] = sum of squares of 1 ... k
static COUNTSUMLIST Row1[MAX_ORDER*MAX_ORDER+1];  // fixed storage for row 1 records
static MINMAX **sRange;        // precomputed s range for updates

static COUNTSUMLIST ***L;      // allocated table of pointers to lists

static int N, emax, S1, S2, e, e2;

static bigint count_bimagic_series(void);
static void precompute_s_ranges(void);
static void null_table_entries(void);
static void update_row_N(void);
static void update_mid_rows(void);
static void update_row_1(void);
static COUNTSUMLIST *merge_lists(COUNTSUMLIST *list1, COUNTSUMLIST *list2, int min2, int max2);
static COUNTSUMLIST *copy_list(COUNTSUMLIST *list);
```

```
//-----------------------------------------------------------
void main(int argc, char **argv)
{
  int r, k;
  bigint total;

  N = atoi(argv[1]);              // read order from command line
  emax = N*N;                     // compute maximum element
  S1 = N*(N*N+1)/2;               // compute magic constant
  S2 = N*(N*N+1)*(2*N*N+1) / 6;   // compute bimagic constant

  L = (COUNTSUMLIST ***)malloc((N+1) * sizeof(COUNTSUMLIST **));
  for (r = 1; r <= N; ++r)
    L[r] = (COUNTSUMLIST **)malloc((S1+1) * sizeof(COUNTSUMLIST *));

  sRange = (MINMAX **)malloc((N+1) * sizeof(MINMAX *));
  for (r = 1; r <= N; ++r)
    sRange[r] = (MINMAX *)malloc((emax+1) * sizeof(MINMAX));

  StartAllocPtr = malloc(RESTOFMEMORY);
  EndAllocPtr = (char *)StartAllocPtr + RESTOFMEMORY;
  NextAllocPtr = StartAllocPtr;

  T[0] = 0;                  // compute T[k] = sum of 1 ... k
  for (k = 1; k <= N; ++k)
    T[k] = T[k-1] + k;

  T2[0] = 0;                 // compute T2[k] = sum of squares of 1 ... k
  for (k = 1; k < emax; ++k)
    T2[k] = T2[k-1] + k*k;

  total = count_bimagic_series();

  free(StartAllocPtr);

  for (r = N; r >= 1; --r)
    free(sRange[r];
  free(sRange);

  for (r = N; r >= 1; --r)
    free(L[r]);
  free(L);

  printf("total = ");
  print_bigint(total);
}
```

```
//-------------------------------------------------------
static bigint count_bimagic_series(void)
{
  precompute_s_ranges();

  null_table_entries();

  e = 1; e2 = 1;
  update_row_1();

  for (e = 2; e <= emax; ++e)
  {
     e2 = e*e;

     update_row_N();
     update_mid_rows();
     update_row_1();
  }

  if (L[N][S1] != NULL)
    return L[N][S1]->csl_record[0].cs_count;

  return 0;  // no bimagic series found
}
//-----------------------------------------------------
// For each stage of each row,
// determine the s range for doing updates from that row.
// This is used to make sure that needed lists are copied
// and not overwritten when the circular buffer wraps around.
//-----------------------------------------------------
static void precompute_s_ranges(void)
{
  int r, k;
  int sminmin, smaxmax;
  int smin, smin1, smin2;
  int smax, smax1, smax2;

  for (r = 3; r <= N-1; ++r)
  { sminmin = 0x7FFFFFFF;
    smaxmax = 0;
    for (e = emax-(N-r); e >= r; --e)
    { k = N - r;
      smin1 = T[r-1];
      smax1 = (r-1)*e - T[r-1];
      smin2 = S1 - k*emax + T[k-1] - e;
      smax2 = S1 - (k+1)*e - T[k];
      smin = max(smin1, smin2);
      smax = min(smax1, smax2);
      sminmin = min(sminmin, smin);
      smaxmax = max(smaxmax, smax);
      sRange[r-1][e-1].min = sminmin;
      sRange[r-1][e-1].max = smaxmax;
    }
  }
```

```
  r = N;
  sminmin = 0x7FFFFFFF;
  smaxmax = 0;
  for (e = emax; e >= N; --e)
  {  smin = smax = S1 - e;
     sminmin = min(sminmin, smin);
     smaxmax = max(smaxmax, smax);
     sRange[r-1][e-1].min = sminmin;
     sRange[r-1][e-1].max = smaxmax;
  }
}
//-----------------------------------------------------------------------
static void null_table_entries(void)
{
  int r, s;

  for (r = 1; r <= N; ++r)
    for (s = 1; s <= S1; ++s)
      L[r][s] = NULL;
}
//-----------------------------------------------------------------------------
static void update_row_N(void)
{
    if (e >= N)
      L[N][S1] = merge_lists(L[N][S1], L[N-1][S1-e], S2, S2);
}
//-----------------------------------------------------------------------------
static void update_mid_rows(void)
{
  int r, s, k;
  int rmin, rmax;
  int smin, smin1, smin2;
  int smax, smax1, smax2;
  int sum2min, sum2max;

  rmax = min(N-1, e);
  rmin = max(2, N-(emax-e));

  k = N - rmax;
  smin1 = e + T[rmax-1];
  smax1 = rmax*e - T[rmax-1];
  smin2 = S1 - k*emax + T[k-1];
  smax2 = S1 - k*e - T[k];

  sum2min = S2 - (T2[emax] - T2[emax-k]);
  sum2max = S2 - (T2[e+k] - T2[e]);
```

```
  for (r = rmax; r >= rmin; --r)
  { smin = max(smin1, smin2);
    smax = min(smax1, smax2);

    if (smin <= smax)
    {
      for (s = sRange[r][e].mm_min; s < smin; ++s)
        L[r][s] = copy_list(L[r][s]);

      for (s = smin; s <= smax; ++s)
      { int max2, ss;
        ss = S1 - s;
        ss = ss*ss;
        max2 = min(S2 - ss / k, sum2max);
        L[r][s] = merge_lists(L[r][s], L[r-1][s-e], sum2min, max2);
      }

      for (s = smax+1; s <= sRange[r][e].mm_max; ++s)
        L[r][s] = copy_list(L[r][s]);
    }
    else
    {
      for (s = sRange[r][e].mm_min; s <= sRange[r][e].mm_max; ++s)
        L[r][s] = copy_list(L[r][s]);
    }

    sum2max += T2[e+k] - T2[e+k+1];
    sum2min += T2[emax-k-1] - T2[emax-k];

    smin2 -= emax - k;
    ++k;
    smin1 -= r - 1;
    smax1 += r - 1 - e;
    smax2 -= e + k;
  }
}

//-------------------------------------------------------------------------
static void update_row_1(void)
{
    if (e <= emax-(N-1))
    { Row1[e].csl_length = 1;
      Row1[e].csl_record[0].cs_count = 1;
      Row1[e].csl_record[0].cs_sum2 = e2;
      L[1][e] = &Row1[e];
    }
}
```

```
//-----------------------------------------------------------------------------
static COUNTSUMLIST *copy_list(COUNTSUMLIST *list)
{
  int cnt;
  COUNTSUMLIST *dstlist;
  COUNTSUM *src, *dst;

  if (list == NULL) return NULL;

  cnt = list->csl_length;

  if (EndAllocPtr - NextAllocPtr < cnt * sizeof(COUNTSUM) + sizeof(int))
    NextAllocPtr = StartAllocPtr;

  dstlist = NextAllocPtr;
  dstlist->csl_length = cnt;
  dst = &distlist->csl_record[0];
  src = &list->csl_record[0];

  while (cnt--)
    *dst++ = *src++;

  NextAllocPtr = dst;

  return dstlist;
}
//-----------------------------------------------------------------------------
static COUNTSUMLIST *merge_lists(COUNTSUMLIST *list1, COUNTSUMLIST *list2,
                                 int min2, int max2)
{
  int cnt1, cnt2;
  int src2sum2e2;
  COUNTSUMLIST *dstlist;
  COUNTSUM *src1, *src2, *dst;

  if (list2 == NULL)
  {  if (list1 == NULL) return NULL;
     cnt2 = 0;
     cnt1 = list1->csl_length;
     src1 = &list1->csl_record[0];
  }
  else if (list1 == NULL)
  {  cnt1 = 0;
     cnt2 = list2->csl_length;
     src2 = &list2->csl_record[0];
     src2sum2e2 = src2->cs_sum2 + e2;
  }
  else
  {  cnt1 = list1->csl_length;
     cnt2 = list2->csl_length;
     src1 = &list1->csl_record[0];
     src2 = &list2->csl_record[0];
     src2sum2e2 = src2->cs_sum2 + e2;
  }
```

```
  if (EndAllocPtr - NextAllocPtr < (cnt1 + cnt2) * sizeof(COUNTSUM) + sizeof(int))
    NextAllocPtr = StartAllocPtr;

  dstlist = (COUNTSUMLIST *)NextAllocPtr;
  dstlist->csl_length = 0;
  dst = &dstlist->csl_record[0];

  while (cnt1 > 0 && cnt2 > 0)
  {
    if (src1->cs_sum2 < src2sum2e2)
    {
      if (src1->cs_sum2 > max2) goto end1;
      if (src1->cs_sum2 >= min2)
      { *dst++ = *src1;
        ++dstlist->csl_length;
      }
      ++src1;
      --cnt1;
    }
    else if (src1->cs_sum2 > src2sum2e2)
    {
      if (src2sum2e2 > max2) goto end1;
      if (src2sum2e2 >= min2)
      { *dst = *src2;
        ++dstlist->csl_length;
        dst->cs_count = src2sum2e2;
        ++dst;
      }
      ++src2;
      if (--cnt2)
        src2sum2e2 = src2->cs_sum2 + e2;
    }
    else
    {
      if (src2sum2e2 > max2) goto end1;
      if (src2sum2e2 >= min2)
      { *dst = *src1;
        ++dstlist->csl_length;
        dst->cs_count += src2->cs_count;     // accumulate counts
        ++dst;
      }
      ++src1;
      ++src2;
      --cnt1;
      if (--cnt2)
        src2sum2e2 = src2->cs_sum2 + e2;
    }
  }
```

```
  while (cnt1--)
  { if (src1->cs_sum2 > max2) goto end1;
    if (src1->cs_sum2 >= min2)
    { *dst++ = *src1;
      ++dstlist->csl_length;
    }
    ++src1;
  }

  while (cnt2--)
  { src2sum2e2 = src2->cs_sum2 + e2;
    if (src2sum2e2 > max2) goto end1;
    if (src2sum2e2 >= min2)
    { dst->cs_count = src2->cs_count;
      dst->cs_sum2 = src2sum2e2;
      ++dst;
      ++dstlist->csl_length;
    }
    ++src2;
  }

end1:
  if (dstlist->csl_length == 0)
  { NextAllocPtr = dstlist;
    return NULL;
  }

  NextAllocPtr = dst;
  return dstlist;
}
//================================================================
```