

**1. Intro.** Given  $m$ ,  $n$ ,  $t$ , and  $z$ , I calculate the  $z$ th matrix with the property that  $0 \leq a_{i,j} < t$  for  $0 \leq i < m$  and  $0 \leq j < n$  and whose histoscape is a three-valent polyhedron. (It's based on the program HISTOSCAPE-COUNT, which simply counts the total number of solutions.)

That program enumerated solutions by dynamic programming, using  $(m-1)(n-1)t^{n+1}$  updates to a huge auxiliary matrix. If I could run those updates backwards, it would be easy to figure out the  $z$ th solution. But I don't want to store all of that information. So I regenerate the auxiliary matrix  $(m-1)(n-1)$  times, taking back the updates one by one. (Eventually this gets easier.)

```
#define maxn 10
#define maxt 16
#define o mems++
#define oo mems += 2
#define ooo mems += 3
#include <stdio.h>
#include <stdlib.h>
int m, n, t; /* command-line parameters */
unsigned long long z; /* another command-line parameter */
char bad[maxt][maxt][maxt]; /* is a submatrix bad? */
unsigned long long *count; /* the big array of counts */
unsigned long long newcount[maxt]; /* counts that will replace old ones */
int firstknown; /* where the good information begins in sol */
unsigned long long mems; /* memory references to octabytes */
int inx[maxn + 1]; /* indices being looped over */
int tpow[maxn + 2]; /* powers of t */
int pos[maxn + 1]; /* what solution position corresponds to each index */
int sol[maxn * maxn]; /* the partial solution known so far */
main(int argc, char *argv[])
{
    register int a, b, c, d, i, j, k, p, q, r, pp, p0;
    <Process the command line 2>;
    <Compute the bad table 3>;
    firstknown = m * n; /* nothing is known at the beginning */
loop: while (firstknown) {
    for (i = 1; i < m; i++)
        for (j = 1; j < n; j++) <Handle constraint (i,j); update the partial solution and goto loop, if
                                we're ready to do that 7>;
    <Set up the first partial solution 5>;
}
    <Print the solution 4>;
}
```

```

2.  ⟨ Process the command line 2 ⟩ ≡
    if (argc ≠ 5 ∨ sscanf(argv[1], "%d", &m) ≠ 1 ∨ sscanf(argv[2], "%d", &n) ≠ 1 ∨ sscanf(argv[3], "%d",
        &t) ≠ 1 ∨ sscanf(argv[4], "%lld", &z) ≠ 1) {
        fprintf(stderr, "Usage: %s m n t z\n", argv[0]);
        exit(-1);
    }
    if (m < 2 ∨ m > maxn ∨ n < 2 ∨ n > maxn) {
        fprintf(stderr, "Sorry, m and n should be between 2 and %d!\n", maxn);
        exit(-2);
    }
    if (t < 2 ∨ t > maxt) {
        fprintf(stderr, "Sorry, t should be between 2 and %d!\n", maxt);
        exit(-3);
    }
    for (j = 1, k = 0; k ≤ n + 1; k++) tpow[k] = j, j *= t;
    count = (unsigned long long *) malloc(tpow[n + 1] * sizeof(unsigned long long));
    if (¬count) {
        fprintf(stderr, "I couldn't allocate t^%d=%d entries for the counts!\n", n + 1, tpow[n + 1]);
        exit(-4);
    }
}

```

This code is used in section 1.

```

3.  ⟨ Compute the bad table 3 ⟩ ≡
    for (a = 0; a < t; a++)
        for (b = 0; b ≤ a; b++)
            for (c = 0; c ≤ b; c++)
                for (d = 0; d ≤ a; d++) {
                    if (d > b) goto nogood;
                    if (a > b ∧ c > d) goto nogood;
                    if (a > b ∧ b ≡ d ∧ d > c) goto nogood;
                    continue;
                }
    nogood: bad[a][b][c][d] = 1;
    bad[a][c][b][d] = 1;
    bad[b][d][a][c] = 1;
    bad[b][a][d][c] = 1;
    bad[d][c][b][a] = 1;
    bad[d][b][c][a] = 1;
    bad[c][a][d][b] = 1;
    bad[c][d][a][b] = 1;
}

```

This code is used in section 1.

```

4.  ⟨ Print the solution 4 ⟩ ≡
    fprintf(stderr, "Solution completed after %lld mems:\n", mems);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) printf("%d", sol[i * n + j]);
        printf("\n");
    }
}

```

This code is used in section 1.

**5.** At this point we’ve done all the computations of HISTOSCAPE-COUNT, essentially without change. In other words, we’ve finished processing the final constraint  $(m-1, n-1)$ , and the *count* table tells us how many solutions have a given setting of the bottom row, as well as a given setting of cell  $(m-2, n-1)$ .

⟨ Set up the first partial solution 5 ⟩  $\equiv$

```

for ( $k = 0$ ;  $k \leq n$ ;  $k++$ ) {
     $o, pos[q] = --firstknown$ ;
    if ( $q \equiv 0$ )  $q = n$ ; else  $q--$ ;
}
for ( $p = 0$ ;  $p < tpow[n+1]$ ;  $p++$ ) {
    if ( $o, z < count[p]$ ) break;
     $z -= count[p]$ ;
}
if ( $p \equiv tpow[n+1]$ ) {
     $fprintf(stderr, "Oops, \_z\_exceeds\_the\_total\_number\_of\_solutions!\n");$ 
     $exit(-4)$ ;
}
for ( $k = 0$ ;  $k \leq n$ ;  $k++$ ) {
     $sol[pos[k]] = p \% t$ ;
     $fprintf(stderr, "cell\_d, \_d\_is\_d\n", pos[k]/n, pos[k] \% n, sol[pos[k]])$ ;
     $p /= t$ ;
}
 $fprintf(stderr, "z\_reset\_to\_lld\n", z)$ ;

```

This code is used in section 1.

**6.** Throughout the main computation, I’ll keep the value of  $p$  equal to  $(inx[n] \dots inx[1] inx[0])_t$ .

Elements of the *pos* array represent cells in the matrix; cell  $(i, j)$  corresponds to the number  $i * n + j$ . When  $inx[r]$  corresponds to a known part of the solution, we “freeze” it.

⟨ Increase the *inx* table, keeping  $inx[q]$  constant 6 ⟩  $\equiv$

```

for ( $r = 0$ ;  $r \leq n$ ;  $r++$ )
    if ( $r \neq q \wedge (o, pos[r] \equiv 0)$ ) {
         $ooo, inx[r]++, p += tpow[r]$ ;
        if ( $inx[r] < t$ ) break;
         $oo, inx[r] = 0, p -= tpow[r+1]$ ;
    }

```

This code is used in sections 7 and 10.

7. Here's the heart of the computation (the inner loop).

One can show that  $q \equiv j - i$  (modulo  $n + 1$ ) when we're working on constraint  $(i, j)$ .

```

⟨ Handle constraint  $(i, j)$ ; update the partial solution and goto loop, if we're ready to do that 7 ⟩ ≡
{
  if  $(j \equiv 1)$  ⟨ Get set to handle constraint  $(i, 1)$  10 ⟩
  else  $q = (q \equiv n ? 0 : q + 1)$ ;
  while (1) {
     $o, b = (q \equiv n ? \text{inx}[0] : \text{inx}[q + 1])$ ;
     $o, c = (q \equiv 0 ? \text{inx}[n] : \text{inx}[q - 1])$ ;
    if  $(i * n + j \geq \text{firstknown})$  ⟨ Work with a known value of  $d$ , possibly making a breakthrough 8 ⟩
    else {
      for  $(d = 0; d < t; d++)$   $o, \text{newcount}[d] = 0$ ;
      for  $(o, a = 0, pp = p; a < t; a++, pp += \text{tpow}[q])$  {
        for  $(d = 0; d < t; d++)$ 
          if  $(o, \neg \text{bad}[a][b][c][d])$   $ooo, \text{newcount}[d] += \text{count}[pp]$ ;
      }
      for  $(o, d = 0, pp = p; d < t; d++, pp += \text{tpow}[q])$   $oo, \text{count}[pp] = \text{newcount}[d]$ ;
    }
    ⟨ Increase the inx table, keeping  $\text{inx}[q]$  constant 6 ⟩;
    if  $(p \equiv p0)$  break;
  }
  if  $(i * n + j \geq \text{firstknown})$   $ooo, \text{pos}[q] = i * n + 1, \text{inx}[q] = \text{sol}[i * n + j], p += \text{inx}[q] * \text{tpow}[q], p0 = p$ ;
   $\text{fprintf}(\text{stderr}, "\text{done with } \%d, \%d. .\%lld, \%lld \text{ mems} \backslash \text{n}", i, j, \text{count}[0], \text{mems})$ ;
}

```

This code is used in section 1.

8. ⟨ Work with a known value of  $d$ , possibly making a breakthrough 8 ⟩ ≡

```

{
   $d = \text{sol}[i * n + j]$ ;
  if  $(i * n + j \equiv \text{firstknown} + n)$  ⟨ Deduce cell  $(i - 1, j - 1)$  and goto loop 9 ⟩;
  for  $(oo, \text{newcount}[d] = 0, a = 0, pp = p; a < t; a++, pp += \text{tpow}[q])$  {
    if  $(o, \neg \text{bad}[a][b][c][d])$   $ooo, \text{newcount}[d] += \text{count}[pp]$ ;
  }
   $o, \text{count}[p + d * \text{tpow}[q]] = \text{newcount}[d]$ ;
}

```

This code is used in section 7.

9.  $\langle \text{Deduce cell } (i-1, j-1) \text{ and } \mathbf{goto} \text{ loop } 9 \rangle \equiv$

```

{
  for ( $o, a = 0, pp = p; a < t; a++, pp += tpow[q]$ )
    if ( $o, \neg bad[a][b][c][d]$ ) {
      if ( $o, z < count[pp]$ ) break;
       $z -= count[pp]$ ;
    }
  if ( $a \equiv t$ ) {
    fprintf(stderr, "internal_error, z too large at %d, %d\n", i, j);
    exit(-6);
  }
   $sol[---firstknown] = a$ ;
  fprintf(stderr, "cell %d, %d is %d; z reset to %lld\n", firstknown/n, firstknown % n, a, z);
  goto loop;
}

```

This code is used in section 8.

10. And here's the tricky part that keeps the inner loop easy. I don't know a good way to explain it, except to say that a hand simulation will reveal all.

$\langle \text{Get set to handle constraint } (i, 1) \text{ } 10 \rangle \equiv$

```

{
  if ( $i \equiv 1$ ) {
     $o, p = q = 0, newcount[0] = 1$ ;
    for ( $r = 0; r \leq n; r++$ ) {
      if ( $r < firstknown$ )  $ooo, pos[r] = inx[r] = 0$ ;
      else  $ooo, pos[r] = r, inx[r] = sol[r], p += inx[r] * tpow[r]$ ;
    }
     $p0 = p$ ;
    while (1) {
      for ( $a = 0, pp = p; a < t; a++, pp += tpow[q]$ )  $o, count[pp] = newcount[0]$ ;
       $\langle \text{Increase the } inx \text{ table, keeping } inx[q] \text{ constant } 6 \rangle$ ;
      if ( $p \equiv p0$ ) break;
    }
  } else {
     $q = (q \equiv n ? 0 : q + 1)$ ;
    if ( $n * i \equiv firstknown + n$ )  $\langle \text{Deduce cell } (i-2, n-1) \text{ and } \mathbf{goto} \text{ loop } 11 \rangle$ ;
    while (1) {
      for ( $o, a = 0, pp = p, newcount[0] = 0; a < t; a++, pp += tpow[q]$ )  $o, newcount[0] += count[pp]$ ;
      if ( $n * i \geq firstknown$ )  $o, count[p + sol[n * i] * tpow[q]] = newcount[0]$ ;
      else for ( $a = 0, pp = p; a < t; a++, pp += tpow[q]$ )  $o, count[pp] = newcount[0]$ ;
       $\langle \text{Increase the } inx \text{ table, keeping } inx[q] \text{ constant } 6 \rangle$ ;
      if ( $p \equiv p0$ ) break;
    }
    if ( $i * n \geq firstknown$ )  $ooo, pos[q] = i * n, inx[q] = sol[i * n], p += inx[q] * tpow[q], p0 = p$ ;
     $q = (q \equiv n ? 0 : q + 1)$ ;
  }
}

```

This code is used in section 7.

11.  $\langle \text{Deduce cell } (i - 2, n - 1) \text{ and } \mathbf{goto} \text{ loop } 11 \rangle \equiv$

```

{
  for ( $o, a = 0, pp = p; a < t; a++, pp += tpow[q]$ ) {
    if ( $o, z < count[pp]$ ) break;
     $z -= count[pp]$ ;
  }
  if ( $a \equiv t$ ) {
    fprintf(stderr, "internal_error, z_too_large_at_%d, 0\n", i);
    exit(-6);
  }
   $sol[--firstknown] = a$ ;
  fprintf(stderr, "cell_%d,%d_is_%d; z_reset_to_%lld\n", i - 2, n - 1, a, z);
  goto loop;
}

```

This code is used in section 10.

**12. Index.**

*a*: [1](#).  
*argc*: [1](#), [2](#).  
*argv*: [1](#), [2](#).  
*b*: [1](#).  
*bad*: [1](#), [3](#), [7](#), [8](#), [9](#).  
*c*: [1](#).  
*count*: [1](#), [2](#), [5](#), [7](#), [8](#), [9](#), [10](#), [11](#).  
*d*: [1](#).  
*exit*: [2](#), [5](#), [9](#), [11](#).  
*firstknown*: [1](#), [5](#), [7](#), [8](#), [9](#), [10](#), [11](#).  
*fprintf*: [2](#), [4](#), [5](#), [7](#), [9](#), [11](#).  
*i*: [1](#).  
*inx*: [1](#), [6](#), [7](#), [10](#).  
*j*: [1](#).  
*k*: [1](#).  
*loop*: [1](#), [9](#), [11](#).  
*m*: [1](#).  
*main*: [1](#).  
*malloc*: [2](#).  
*maxn*: [1](#), [2](#).  
*maxt*: [1](#), [2](#).  
*mems*: [1](#), [4](#), [7](#).  
*n*: [1](#).  
*newcount*: [1](#), [7](#), [8](#), [10](#).  
*nogood*: [3](#).  
*o*: [1](#).  
*oo*: [1](#), [6](#), [7](#), [8](#).  
*ooo*: [1](#), [6](#), [7](#), [8](#), [10](#).  
*p*: [1](#).  
*pos*: [1](#), [5](#), [6](#), [7](#), [10](#).  
*pp*: [1](#), [7](#), [8](#), [9](#), [10](#), [11](#).  
*printf*: [4](#).  
*p0*: [1](#), [7](#), [10](#).  
*q*: [1](#).  
*r*: [1](#).  
*sol*: [1](#), [4](#), [5](#), [7](#), [8](#), [9](#), [10](#), [11](#).  
*sscanf*: [2](#).  
*stderr*: [2](#), [4](#), [5](#), [7](#), [9](#), [11](#).  
*t*: [1](#).  
*tpow*: [1](#), [2](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#).  
*z*: [1](#).

- ⟨ Compute the *bad* table 3 ⟩ Used in section 1.
- ⟨ Deduce cell  $(i - 1, j - 1)$  and **goto loop** 9 ⟩ Used in section 8.
- ⟨ Deduce cell  $(i - 2, n - 1)$  and **goto loop** 11 ⟩ Used in section 10.
- ⟨ Get set to handle constraint  $(i, 1)$  10 ⟩ Used in section 7.
- ⟨ Handle constraint  $(i, j)$ ; update the partial solution and **goto loop**, if we're ready to do that 7 ⟩ Used in section 1.
- ⟨ Increase the *inx* table, keeping *inx*[*q*] constant 6 ⟩ Used in sections 7 and 10.
- ⟨ Print the solution 4 ⟩ Used in section 1.
- ⟨ Process the command line 2 ⟩ Used in section 1.
- ⟨ Set up the first partial solution 5 ⟩ Used in section 1.
- ⟨ Work with a known value of *d*, possibly making a breakthrough 8 ⟩ Used in section 7.



# HISTOSCAPE-UNRANK

	Section	Page
Intro .....	<a href="#">1</a>	1
Index .....	<a href="#">12</a>	7