(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. This program, inspired by HISTOSCAPE-COUNT, calculates the number of $m \times n$ "whirlpool permutations," given m and n.

What's a whirlpool permutation, you ask? Good question. An $m \times n$ matrix has $(m-1)(n-1)$ submatrices of size 2×2. An $m \times n$ whirlpool permutation is a permutation of $(mn)!$ elements for which the relative order of the elements in each of those submatrices is a "vortex"—that is, it travels a cyclic path from smallest to largest, either clockwise or counterclockwise.

Thus there are exactly eight 2×2 whirlpool permutations. If the entries of the matrix are denoted *abcd* from top to bottom and left to right, they are 1243, 1423, 2134, 2314, 3241, 3421, 4132, 4312. One simple test is to compare $a : b$, $b : d$, $d : c$, $c : a$; the number of '$<$' must be odd. (Hence the number of '$>$' must also be odd.)

The enumeration is by a somewhat mind-boggling variant of dynamic programming that I've not seen before. It needs to represent $n+1$ elements of a permutation of t elements, where t is at most mn, and there are up to $(mn)^{\underline{n+1}}$ such partial permutations; so I can't expect to solve the problem unless m and n are fairly small. Even so, when I *can* solve the problem it's kind of thrilling, because this program makes use of a really interesting way to represent $t^{\underline{n+1}}$ counts in computer memory.

The same method could actually be used to enumerate matrices of permutations whose 2×2 submatrices satisfy any arbitrary relations, when those relations depend only the relative order of the four elements. (Thus any of 2^{24} constraints might be prescribed for each of the $(m-1)(n-1)$ submatrices. The whirlpool case, which accepts only the eight relative orderings listed above, is just one of zillions of possibilities.)

It's better to have $m \ge n$. But I'll try some cases with $m < n$ too, for purposes of testing.

```
#define maxn 8
#define maxmn 36
#define o mems++
#define oo mems += 2
#define ooo mems += 3

#include <stdio.h>
#include <stdlib.h>
  int m, n;     /* command-line parameters */
  unsigned long long *count;     /* the big array of counts */
  unsigned long long newcount[maxmn];     /* counts that will replace old ones */
  unsigned long long mems;     /* memory references to octabytes */
  int x[maxn + 1];     /* indices being looped over */
  int ay[maxn + 1];
  int l[maxmn], u[maxmn];
  int tpow[maxmn + 1];     /* factorial powers t^{n+1} */
  ⟨ Subroutines 4 ⟩;
  main(int argc, char *argv[])
  {
    register int a, b, c, d, i, j, k, p, q, r, mn, t, tt, kk, bb, cc, pdel;
    ⟨ Process the command line 2 ⟩;
    for (i = 1; i < m; i++)
      for (j = 0; j < n; j++) ⟨ Handle constraint (i, j) 8 ⟩;
    ⟨ Print the grand total 9 ⟩;
  }
```

2. ⟨Process the command line 2⟩ ≡

```
if (argc ≠ 3 ∨ sscanf(argv[1], "%d", &m) ≠ 1 ∨ sscanf(argv[2], "%d", &n) ≠ 1) {
  fprintf(stderr, "Usage:␣%s␣m␣n\n", argv[0]);
  exit(−1);
}
mn = m ∗ n;
if (m < 2 ∨ m > maxn ∨ n < 2 ∨ n > maxn ∨ mn > maxmn) {
  fprintf(stderr, "Sorry,␣m␣and␣n␣should␣be␣between␣2␣and␣%d,␣with␣mn<=%d!\n", maxn, maxmn);
  exit(−2);
}
for (k = n + 1; k ≤ mn; k++) {
  register unsigned long long acc = 1;
  for (j = 0; j ≤ n; j++) acc ∗= k − j;
  if (acc ≥ #80000000) {
    fprintf(stderr, "Sorry,␣mn\\falling(n+1)␣must␣be␣less␣than␣2^31!\n");
    exit(−666);
  }
  tpow[k] = acc;
}
count = (unsigned long long ∗) malloc(tpow[mn] ∗ sizeof(unsigned long long));
if (¬count) {
  fprintf(stderr, "I␣couldn't␣allocate␣%d␣entries␣for␣the␣counts!\n", tpow[mn]);
  exit(−4);
}
```

This code is used in section 1.

3. Suppose I want to represent $n+1$ specified elements of a permutation of $t+1$ elements. For example, we might have $n = 3$ and $t = 8$, and the final four elements of a permutation $y_0 \ldots y_8$ might be $y_5y_6y_7y_8 = 3142$. There are $(t+1)^{\underline{n+1}}$ such partial permutations, and I can represent them compactly with $n+1$ integer variables x_{t-n}, …, x_{t-1}, x_t, where $0 \le x_j \le j$. The rule is that x_j is $y_j - w_j$, where w_j is the number of elements "inverted" by y_j (the number of elements to the right of y_j that are less than y_j). In the example, $w_0w_1w_2w_3 = 2010$, so $x_5x_6x_7x_8 = 1132$. (Or going backward, if $x_5x_6x_7x_8 = 3141$ then $y_5y_6y_7y_8 = 6251$.)

This representation has a beautiful property that we shall exploit. Every permutation $y_0 \ldots y_t$ of $\{0, \ldots, t\}$ yields $t+2$ permutations $y'_0 \ldots y'_{t+1}$ of $\{0, \ldots, t+1\}$ if we choose y'_{t+1} arbitrarily, and then set $y'_j = y_j + [y_j \ge y'_{t+1}]$. For example, if $t = 8$ and $y_5y_6y_7y_8 = 3142$, the ten permutations obtained from $y_0 \ldots y_8$ will have $y'_5y'_6y'_7y'_8y'_9 = 42530$, 42531, 41532, 41523, 31524, 31425, 31426, 31427, 31428, or 31429. And the representations $x'_5x'_6x'_7x'_8x'_9$ of those last five elements will simply be respectively 31420, 31421, …, 31429! In general, we'll have $x'_j = x_j$ for $0 \le j \le t$, and $x'_{t+1} = y'_{t+1}$ will be arbitrary.

4. Now comes the mind-boggling part. I want to maintain a count $c(x_{t-n},\ldots,x_t)$ for each setting of the indices $(x_{t-n},\ldots,x_t)$, but I want to put those counts into memory in such a way that I won't clobber any of the existing counts when I'm updating t to $t{+}1$. In particular, if $x'_{t+1}\le t{-}n$, I'll want $c(x'_{t+1-n},\ldots,x'_t,x'_{t+1})$ to be stored in exactly the same place as $c(x'_{t+1},x_{t+1-n},\ldots,x_t)$ was stored in the previous round. But if $x'_{t+1}>t-n$, I'll store $c(x'_{t+1-n},\ldots,x'_t,x'_{t+1})$ in a brand-new, previously unused location of memory.

Thus we shall use a memory mapping function μ_t, different for each t, such that $c(x_{t-n},x_{t-n+1},\ldots,x_t)$ is stored in location $\mu_t(x_{t-n},x_{t-n+1},\ldots,x_t)$ during round t of the computation.

Let's go back to the example in the previous section and apply it to whirlpool permutations for $n=3$. Denote the permutation in the first three rows by $y_0\ldots y_8$, where $y_6y_7y_8$ is the third row and y_5 is the last element of the second row. (It's a permutation of $\{0,\ldots,8\}$, representing the relative order of a final permutation of $\{0,\ldots,3m-1\}$ that will fill the entire matrix.) At this point we've calculated counts $c(x_5,x_6,x_7,x_8)$ that tell us how many such partial whirlpool permutations have any given setting of $y_5y_6y_7y_8$. In particular, $c(1,1,3,2)$ counts those for which $y_5y_6y_7y_8=3142$.

To get to the next round, we essentially want to know how many partial permutations $y'_0\ldots y'_9$ of $\{0,\ldots,9\}$ will have a given setting of $y'_6y'_7y'_8y'_9$; the second row is now irrelevant to future computations. It's the same as asking how many permutations have $y_6y_7y_8=142$. Answer: $c(0,1,3,2)+c(1,1,3,2)+c(2,1,3,2)+c(3,1,3,2)+c(4,1,3,2)+c(5,1,3,2)$, because these count the permutations with $y_5y_6y_7y_8=0142$, 3142, 5142, 6142, 7142, 8142.

Those six counts $c(k,1,3,2)$ appear in memory locations $\mu_8(k,1,3,2)$, for $0\le k\le 5$. On the next round we'll want $c'(x'_6,x'_7,x'_8,x'_9)=c'(1,3,2,x'_9)$ to be set to their sum. These new counts will appear in memory locations $\mu_9(1,3,2,x'_9)$. So we'd like $\mu_9(1,3,2,k)=\mu_8(k,1,3,2)$ when $0\le k\le 5$.

Let $\lambda_t(x_{t-n},\ldots,x_t)=\bigl(\cdots((x_tt+x_{t-1})(t-1)+x_{t-2})\cdots\bigr)(t-n+1)+x_{t-n}=x_tt^{\underline{n}}+x_{t-1}(t-1)^{\underline{n-1}}+\cdots x_{t-n}(t{-}n)^{\underline{0}}$ be the standard mixed-radix representation of $(x_t\ldots x_{t-n})$ with radices $(t{+}1,t,\ldots,t{-}n{+}1)$. When each x_j ranges from 0 to j, $\lambda_t(x_{t-n},\ldots,x_t)$ ranges from $\lambda_t(0,\ldots,0)=0$ to $\lambda_t(t-n,\ldots,t)=(t+1)^{\underline{n+1}}-1$. Therefore λ_t would be the natural choice for μ_t, if we didn't want to avoid clobbering.

Instead, we use λ_t only when x_t is sufficiently large: We define

$$\mu_t(x_{t-n},\ldots,x_t)=\begin{cases}\lambda_t(x_{t-n},\ldots,x_t), & \text{if } x_t\ge t-n;\\ \mu_{t-1}(x_t,x_{t-n},\ldots,x_{t-1}), & \text{if } x_t\le t-n-1.\end{cases}$$

This recursion terminates with $\mu_n=\lambda_n$, because x_n is always ≥ 0. One can also show that $\mu_{n+1}=\lambda_{n+1}$.

Back to our earlier example, what is $\mu_8(k,1,3,2)$? Since $2\le 4$, it's $\mu_7(2,k,1,3)$. And since $3\le 3$, it's $\mu_6(3,2,k,1)$. Which is $\mu_5(1,3,2,k)$. Finally, therefore, if $k\le 1$, the value is $\lambda_4(k,1,3,2)=68+k$; but if $2\le k\le 5$ it's $\lambda_5(1,3,2,k)=60k+34$.

In this program we will keep x_j in location $x_{j \bmod (n+1)}$. Consequently the arguments to μ_t and λ_t will always be in locations $(x_{(t+1)\bmod (n+1)},x_{(t+2)\bmod (n+1)},\ldots,x_{t\bmod (n+1)})$.

⟨ Subroutines 4 ⟩ ≡

```
int mu(int t)
{
  register int r, a, p, tt;
  for (r = t % (n + 1), tt = t; o, x[r] < tt - n; tt--, r = (r ? r - 1 : n)) ;
  for (o, p = x[r], r = (r ? r - 1 : n), a = 0; a < n; a++, r = (r ? r - 1 : n)) o, p = p * (tt - a) + x[r];
  return p;
}
```

This code is used in section 1.

5. A backtrack essentially like Algorithm 7.2.1.2X nicely runs through all combinations of $x_{t-n+1}\ldots x_t$ and $y_{t-n+1}\ldots y_t$ simultaneously, while also providing a linked list that shows the possibilities for y_{t-n} as x_{t-n} varies from 0 to $t-n$.

The algorithm generates all of the "n-variations" of $\{0,\ldots,t\}$, namely all n-tuples $a_0\ldots a_{n-1}$ of distinct integers in that set, where a_j corresponds to y_{t-j} in the discussion above.

```
⟨ Generate the x's and y's 5 ⟩ ≡
x1: for (k = 0; k ≤ t; k++) o, l[k] = k + 1;
  o, l[t + 1] = 0;      /* circularly linked list */
  k = 0, kk = t % (n + 1);
x2: if (k ≡ n) ⟨ Visit a_0 ... a_{n-1} and goto x6 6 ⟩;
  oo, p = t + 1, q = l[p], x[kk] = 0;
x3: o, ay[k] = q;
x4: ooo, u[k] = p, l[p] = l[q], k++, kk = (kk ? kk − 1 : n);
  goto x2;
x5: o, p = q, q = l[p];
  if (q ≤ t) {
    oo, x[kk]++;
    goto x3;
  }
x6: if (−−k ≥ 0) {
    kk = (kk ≡ n ? 0 : kk + 1);
    ooo, p = u[k], q = ay[k], l[p] = q;
    goto x5;
  }
```

This code is used in section 8.

6. At this point we're ready to do the "inner loop" calculation, by using all counts $c(x_{t-n},\ldots,x_t)$ for $0 \le x_{t-n} \le t-n$ to obtain updated counts that will allow us to increase t. The array $a_{n-1}\ldots a_0$ corresponds to $y_{t-n+1}\ldots y_t$ in the discussion above; we want to loop over all choices for y_{t-n}, namely all choices for a_n. Fortunately there's a linked list containing precisely those choices, beginning at $l[t+1]$.

```
⟨Visit a₀ … aₙ₋₁ and goto x6 6⟩ ≡
  {
    ⟨If possible, find p and pdel so that c(xₜ₋ₙ, …, xₜ) is count[p + pdel ∗ x[kk]] 7⟩;
    for (d = 0; d ≤ t + 1; d++) o, newcount[d] = 0;
    oo, b = ay[n − 1], c = ay[0];
    if (b < c) bb = b, cc = c;
    else bb = c, cc = b;      /∗ min and max ∗/
    {
      register unsigned long long tmp;
      for (oo, a = l[t + 1], x[kk] = 0; a ≤ t; oo, a = l[a], x[kk]++) {
        if (pdel) tmp = count[p + x[kk] ∗ pdel];
        else tmp = count[mu(t − n)];      /∗ if pdel = 0 then mu(t) = mu(t − n) ∗/
        if (j ≡ 0) newcount[0] += tmp;      /∗ no constraint, beginning a new row ∗/
        else if (a < bb ∨ a > cc) {      /∗ whirlpool constraint when a not middle ∗/
          for (d = bb + 1; d ≤ cc; d++) oo, newcount[d] += tmp;
        } else {      /∗ whirlpool constraint when d not middle ∗/
          for (d = 0; d ≤ bb; d++) oo, newcount[d] += tmp;
          for (d = cc + 1; d ≤ t + 1; d++) oo, newcount[d] += tmp;
        }
      }
      if (pdel) {
        for (d = 0; d ≤ t − n; d++) oo, count[p + d ∗ pdel] = newcount[j ? d : 0];
        for ( ; d ≤ t + 1; d++) ooo, x[kk] = d, count[mu(t + 1)] = newcount[j ? d : 0];
      } else {
        for (d = 0; d ≤ t + 1; d++) ooo, x[kk] = d, count[mu(t + 1)] = newcount[j ? d : 0];
      }
    }
    goto x6;
  }
```

This code is used in section 5.

7. Our example of $\mu_8(k,1,3,2)$ shows that the mission of this step is sometimes impossible. But the addressing scheme is usually simple, so I decided to exploit that fact. (Being aware, of course, that premature optimization is the root of all evil in programming.)

```
⟨If possible, find p and pdel so that c(xₜ₋ₙ, …, xₜ) is count[p + pdel ∗ x[kk]] 7⟩ ≡
  for (tt = t, a = 0, r = t % (n + 1); a < n; a++, tt−−, r = (r ? r − 1 : n))
    if (o, x[r] ≥ tt − n) break;
  if (a ≡ n) pdel = 0;      /∗ a difficult case ∗/
  else {
    for (p = pdel = 0, a = 0; a ≤ n; a++, r = (r ? r − 1 : n)) {
      if (r ≠ kk) p = p ∗ (tt + 1 − a) + x[r], pdel = pdel ∗ (tt + 1 − a);
      else p = p ∗ (tt + 1 − a), pdel = pdel ∗ (tt + 1 − a) + 1;
    }
  }
```

This code is used in section 6.

8. ⟨ Handle constraint (i, j) 8 ⟩ ≡

```
{
  t = n * i + j − 1;
  if (t < n) {
    for (p = 0; p < tpow[n + 1]; p++) o, count[p] = 1;
    continue;
  }
  ⟨ Generate the x's and y's 5 ⟩;
  fprintf(stderr, "␣done␣with␣%d,%d␣..%lld,␣%lld␣mems\n", i, j, count[0], mems);
}
```

This code is used in section 1.

9. **#define** *thresh* 1000000000000000000

⟨ Print the grand total 9 ⟩ ≡

```
for (newcount[0] = newcount[1] = newcount[2] = 0, p = tpow[mn] − 1; p ≥ 0; p−−) {
  if (count[p] > newcount[2]) newcount[2] = count[p], pdel = p;
  o, newcount[0] += count[p];
  if (newcount[0] ≥ thresh) ooo, newcount[0] −= thresh, newcount[1]++;
}
fprintf(stderr, "(Maximum␣count␣%lld␣is␣obtained␣for␣params", newcount[2]));
for (q = mn − n − 1; q < mn; q++) {
  fprintf(stderr), "␣%d", pdel % (q + 1));
  pdel /= q + 1;
}
fprintf(stderr, ")\n"();
if (newcount[1] ≡ 0)
  printf("Altogether␣%lld␣%dx%d␣whirlpool␣perms␣(%lld␣mems).\n", newcount[0], m, n, mems);
else printf("Altogether␣%lld%018lld␣%dx%d␣whirlpool␣perms␣(%lld␣mems).\n", newcount[1],
      newcount[0], m, n, mems);
```

This code is used in section 1.

10. Index.

⟨ Generate the x's and y's 5 ⟩ Used in section 8.
⟨ Handle constraint (i,j) 8 ⟩ Used in section 1.
⟨ If possible, find p and *pdel* so that $c(x_{t-n},\ldots,x_t)$ is $count[p + pdel * x[kk]]$ 7 ⟩ Used in section 6.
⟨ Print the grand total 9 ⟩ Used in section 1.
⟨ Process the command line 2 ⟩ Used in section 1.
⟨ Subroutines 4 ⟩ Used in section 1.
⟨ Visit $a_0 \ldots a_{n-1}$ and **goto** *x6* 6 ⟩ Used in section 5.

WHIRLPOOL-COUNT