1. Intro. This little program finds the parade of rank r from among the $B_{m,n}$ parades that can be made by m girls and n boys, given m, n, and r, using the alternative ranking scheme in section 5 of my unpublication "Poly-Bernoulli Bijections."

Apology: I hacked this *very* hastily. It is *not* robust: It doesn't check for overflow, if the numbers exceed 63 bits.

```
#define maxn 25
#include <stdio.h>
#include <stdlib.h>
  int m, n;      /* command-line parameters */
  long long r;      /* command-line parameter */
  long long pB[maxn][maxn];      /* poly-Bernoulli numbers */
  int bico[maxn][maxn];      /* binomial coefficients */
  int gsg[maxn + 1], gsb[maxn + 1];      /* growth sequences for girls, boys */
  int ord;      /* order of the global parade */
  int samp[maxn];      /* subset to be recursively collapsed */
  ⟨Subroutines 5⟩;
  main(int argc, char *argv[])
  {
    register int i, j, k, kk;
    register long long f, s, t;
    register double ff, ss, tt;
    ⟨Compute the bico table 2⟩;
    ⟨Initialize the pB table 4⟩;
    ⟨Process the command line 3⟩;
    unrank(m, n, r, stdout);
  }
```

2. ⟨Compute the *bico* table 2⟩ ≡

```
  for (j = 0; j < maxn; j++) bico[j][0] = 1;
  for (j = 1; j < maxn; j++)
    for (i = 1; i ≤ j; i++) bico[j][i] = bico[j − 1][i] + bico[j − 1][i − 1];
```

This code is used in section 1.

3. ⟨Process the command line 3⟩ ≡

```
  if (argc ≠ 4 ∨ sscanf(argv[1], "%d", &m) ≠ 1 ∨ sscanf(argv[2], "%d", &n) ≠ 1 ∨ sscanf(argv[3], "%lld",
        &r) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣m␣n␣r\n", argv[0]);
    exit(−1);
  }
  if (m ≥ maxn ∨ n ≥ maxn) {
    fprintf(stderr, "Sorry,␣m␣and␣n␣must␣be␣less␣than␣%d!\n", maxn);
    exit(−2);
  }
```

This code is used in section 1.

4. We compute pB numbers only as needed: $pB[m][n]$ is negative if $B_{m,n}$ hasn't yet been computed; otherwise it's a “cache memo” of the true value.

```
⟨ Initialize the pB table 4 ⟩ ≡
  for (j = 0; j < maxn; j++) pB[0][j] = pB[j][0] = 1;
  for (i = 1; i < maxn; i++)
    for (j = 1; j < maxn; j++) pB[i][j] = −1;
```

This code is used in section 1.

5. Here's a subroutine that produces $B_{m,n}$ on demand. It uses the recurrence

$$B_{m+1,n} = B_{m,n} + \sum_{k=1}^{n} \binom{n}{k} B_{m,n+1-k},$$

which is also the basis for the recursive unranking procedure that we are implementing.

```
⟨ Subroutines 5 ⟩ ≡
  long long getpB(int mm, int n)
  {
    register int m, k;
    register long long s;
    if (pB[mm][n] < 0) {
      m = mm − 1;
      s = getpB(m, n);
      for (k = 1; k ≤ n; k++) s += bico[n][k] ∗ getpB(m, n + 1 − k);
      pB[mm][n] = pB[n][mm] = s;
    }
    return pB[mm][n];
  }
```

See also section 6.

This code is used in section 1.

6. And here is that recursive procedure itself. It returns the desired parade in the global arrays *gsg* and *gsb*, which will define ordered partitions of order *ord* for *mm* girls and *n* boys.

```
⟨Subroutines 5⟩ +≡
  void unrank(int mm, int n, long long r, FILE *outfile)
  {
    register int i, j, m, k, kk, l, nn, p, pp, split, r0, max;
    register long long t, rr = r;
    if (mm ≡ 0) ⟨Set up a trivial parade (no girls) 7⟩
    else {
      k = 0, m = mm − 1, t = getpB(m, n);
      if (r < t) unrank(m, n, r, stderr);
      else {
        for (k = 1; ; k++) {
          if (k > n) {
            fprintf(stderr, "r␣is␣too␣big!\n");
            exit(−666);
          }
          r −= t;
          t = bico[n][k] ∗ getpB(m, n + 1 − k);
          if (r < t) break;
        }
        unrank(m, n + 1 − k, r % pB[m][n + 1 − k], stderr);
        r = r/pB[m][n + 1 − k];
        ⟨Unrank the rth k-subset of {1, . . . , n} into samp 13⟩;
      }
      ⟨Build the larger parade by lifting the smaller one via samp 8⟩;
    }
    ⟨Print the parade 12⟩;
  }
```

7. ⟨Set up a trivial parade (no girls) 7⟩ ≡

```
  {
    ord = 0;
    for (i = 1; i ≤ n; i++) gsb[i] = 0;
  }
```

This code is used in section 6.

8. The heart of this program is the "lifting" process, which inverts the mapping $\Pi \mapsto \Pi'$ described in my unpublication.

We're given an ordered partition for m girls into *ord* blocks, in *gsg*; also an ordered partition for $n+1-k$ boys into *ord* blocks, in *gsb*; also a set of $k > 0$ boys, listed in *samp* in increasing order of age. The basic idea is to extend the parade by putting all of *samp* in place of its oldest boy, and to make that sample immediately follow a newly inserted girl $mm = m + 1$.

Suppose the oldest boy in the sample is named Max. He is boy number $samp[k-1]-(k-1)$ before lifting; but he'll be number $samp[k-1]$ afterwards, because we'll renumber *all* boys to agree with *samp*.

Assume that Max is in block p of the given parade. If he's alone in that block, we simply place girl *mm* ahead of him. Otherwise, however, we split him off from the other boys of block p (some of which might be older, some younger), and put girl *mm* between him and his former fellows. That increases *ord*, introduces a new block $p+1$, and causes later block numbers to be stepped up.

One further complication is that we use $p = 0$ to encode the final block of boys, if a boy comes last. Therefore block '$p+1$' has to be properly understood.

```
⟨Build the larger parade by lifting the smaller one via samp 8⟩ ≡
  if (k ≡ 0) ⟨Append a new girl at the end 11⟩
  else {
    nn = n + 1 − k, max = samp[k − 1] − (k − 1), p = gsb[max];
    ⟨If Max isn't alone in block p, set split to 1 10⟩;
    if (split) {    /∗ at least one other boy is also in block p ∗/
      ord++;
      if (p ≡ 0) {    /∗ actually those boys are in the largest block ∗/
        for (j = 1; j ≤ nn; j++)
          if (gsb[j] ≡ 0) gsb[j] = ord;
      }
    }
    ⟨Insert samp into block p 9⟩;
  }
```

This code is used in section 6.

9. Here's the coolest section of this program (but it's tricky, so I hope I've got it right). We can work in place because $j \ge i$ in this loop.

```
⟨Insert samp into block p 9⟩ ≡
  for (i = nn, j = n, l = k − 2; i; i−−, j−−) {
    while (l ≥ 0 ∧ j ≡ samp[l]) l−−, j−−;    /∗ leave holes for the sample ∗/
    pp = gsb[i];
    if (split ∧ p > 0 ∧ pp > p) gsb[j] = pp + 1;
    else gsb[j] = pp;
  }
  for (l = 0; l < k; l++) gsb[samp[l]] = (p ? p + split : 0);    /∗ fill the holes ∗/
  if (split) {
    if (p ≡ 0) gsg[mm] = ord;
    else {
      for (j = 1; j ≤ m; j++)
        if (gsg[j] ≥ p) gsg[j]++;
      gsg[mm] = p;
    }
  } else gsg[mm] = (p ? p − 1 : ord);
```

This code is used in section 8.

10. Max is alone if and only if he's the only guy in block p.

⟨If Max isn't alone in block p, set *split* to 1 10⟩ ≡

```
for (split = −1, j = 1; j ≤ nn; j++)
  if (gsb[j] ≡ p ∧ ++split) break;
```

This code is used in section 8.

11. ⟨Append a new girl at the end 11⟩ ≡

```
{
  fprintf(stderr, "extend␣with␣empty␣set\n");
  for (j = 1; j ≤ n; j++)
    if (gsb[j] ≡ 0) break;
  if (j ≤ n) {      /* appending g_{m+1} after a boy */
    ord++;
    for (j = 1; j ≤ n; j++)
      if (gsb[j] ≡ 0) gsb[j] = ord;
  }
  gsg[mm] = ord;
}
```

This code is used in section 8.

12. ⟨Print the parade 12⟩ ≡

```
fprintf(outfile, "Parade␣%lld␣for␣%d␣and␣%d:", rr, mm, n);
for (j = 0; j ≤ ord; ) {
  for (i = 1; i ≤ mm; i++)
    if (gsg[i] ≡ j) fprintf(outfile, "␣g%d", i);
  j++;
  for (i = 1; i ≤ n; i++)
    if ((j > ord ∧ gsb[i] ≡ 0) ∨ (j ≤ ord ∧ gsb[i] ≡ j)) fprintf(outfile, "␣b%d", i);
}
fprintf(outfile, "\n");
```

This code is used in section 6.

13. Of course we use the recurrence $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$ here.

⟨Unrank the rth k-subset of $\{1, \ldots, n\}$ into *samp* 13⟩ ≡

```
nn = n, kk = k, r0 = r;
while (kk) {
  if (r < bico[nn − 1][kk]) nn−−;
  else r −= bico[nn − 1][kk], samp[−−kk] = nn−−;
}
fprintf(stderr, "extend␣with");
for (l = 0; l < k; l++) fprintf(stderr, "␣b%d", samp[l]);
fprintf(stderr, "␣(sample␣%d␣of␣$%d\\choose%d$)\n", r0, n, k);
```

This code is used in section 6.

14. Index.

UNRANK-PARADE2