1. Intro. This program computes matrix representations of permutations, based on tableaux of a given shape λ. If there are f standard Young tableaux of that shape, it produces $f \times f$ matrices B_π for any given permutation π, with the property that $B_\pi B_\sigma = B_{\pi\sigma}$.

I'm trying to learn concrete details of such representations, and my experience has always been that the best way to learn something is to try to program it whenever possible. Therefore I'm writing this code as part of my own education. But I haven't seen any book that mentions the method used below, so other readers may also find aspects of interest here. Of course I can't claim to have read very much of the huge literature that already exists on this topic; probably I have rediscovered something that's fairly well known.

Let λ be a partition of n, namely $\lambda = (a_1, \ldots, a_k)$ where $a_1 \ge \cdots \ge a_k \ge 1$ and $a_1 + \cdots + a_k = n$. A *tableau* of shape λ is a way to place the numbers $\{1, \ldots, n\}$ into an array with n left-justified rows and a_j entries in row j. The tableau is *standard* if the entries in each row are increasing from left to right and the entries in each column are increasing from top to bottom.

The method of this program is based on a straightforward algorithm that takes a not-necessarily-standard tableau and determines all ways to permute its columns in such a fashion that a subsequent row-sorting will produce a standard tableau. For example, if $\lambda = (3, 3, 3)$ and if the given tableau is

3	1	4
5	9	2
6	8	7

,

there are nine solutions,

3 1 2	3 1 2	5 1 2	6 1 2	3 1 4	3 1 4	3 1 7	5 1 4	6 1 4
5 8 4	6 8 4	3 8 4	3 8 4	5 8 2	6 8 2	5 8 2	6 8 2	5 8 2
6 9 7	5 9 7	6 9 7	5 9 7	6 9 7	5 9 7	6 9 4	3 9 7	3 9 7

;

row-sorting converts these respective solutions to the standard tableaux

1 2 3	1 2 3	1 2 5	1 2 6	1 3 4	1 3 4	1 3 7	1 4 5	1 4 6
4 5 8	4 6 8	3 4 8	3 4 8	2 5 8	2 6 8	2 5 8	2 6 8	2 5 8
6 7 9	5 7 9	6 7 9	5 7 9	6 7 9	5 7 9	4 6 9	3 7 9	3 7 9

.

Another way to state the given problem is, "Find all standard tableaux that can be produced from a given one by permuting columns, then permuting rows."

The first line of standard input should contain the partition elements a_1, ..., a_k, separated by spaces and followed by 0. Subsequent lines should contain permutations whose representative matrices are desired; each permutation is given as a sequence p_1, ..., p_n, separated by spaces.

N.B.: The permutation $p_1 \ldots p_n$ takes $1 \mapsto p_1$, $2 \mapsto p_2$, etc., and the representation matrices produced by this program multiply permutations from left to right. Thus, for example, if A, B, and C are the matrices representing 132, 213, and 231, respectively, aka the permutations (23), (12), and (123), we have $(23)(12) = (123)$ hence $AB = C$.

2. `#define maxn 100 /* let's not permute more than a hundred guys */`
`#define maxf 300 /* and let's not find matrices of size more than 300 × 300 */`

```
#include <stdio.h>
  ⟨ Global variables 3 ⟩
  ⟨ Subroutines 11 ⟩
  main( )
  {
    register int j, jj, k, l;
    ⟨ Read the shape λ 4 ⟩;
    ⟨ Compute the transposed shape λ^T 5 ⟩;
    ⟨ Find all the standard tableaux of the given shape 6 ⟩;
    printf("There␣are␣%d␣standard␣tableaux␣of␣shape", f);
    for (j = 1; j ≤ kk; j++) printf("␣%d", a[j]);
    printf(".\n");
    ⟨ Compute f basis vectors for the representation 16 ⟩;
    while (1) {
      ⟨ Read a permutation p (but break if there's no more good data) 17 ⟩;
      printf("Representation␣of");
      for (j = 1; j ≤ n; j++) printf("␣%d", p[j]);
      printf(":\n");
      for (jj = 0; jj < f; jj++) {
        ⟨ Compute the representation of the jjth standard tableau, permuted by p 18 ⟩;
        ⟨ Reduce the representation to a linear combination of basis elements 19 ⟩;
        for (j = 0; j < f; j++) printf("%␣3d", rep[j]);
        printf("\n");
      }
    }
  }
```

3. ⟨ Global variables 3 ⟩ ≡

```
int a[maxn + 2]; /* the shape */
int b[maxn + 1]; /* its transpose */
int n; /* the number of elements permuted */
int kk; /* the number of rows */
int f; /* the number of standard tableaux */
int p[maxn + 1]; /* the permutation to be represented */
int q[maxn + 1]; /* the inverse of p */
int t[maxn][maxn], tt[maxn][maxn]; /* working tableaux */
int aa[maxn + 2]; /* row sizes of tt */
int stand[maxf][maxn]; /* standard tableaux */
int basis[maxf][maxf]; /* basis elements */
int rep[maxf]; /* a linear combination of standard tableaux */
```

See also section 9.

This code is used in section 2.

4. ⟨ Read the shape λ 4 ⟩ ≡

```
for (j = 0; ; j++) {
  if (j > maxn) {
    fprintf(stderr, "Partition too long (maxn=%d)!\n", maxn);
    exit(−1);
  }
  if (scanf("%d", &a[j + 1]) ≠ 1) {
    fprintf(stderr, "Partition should end with zero!\n");
    exit(−2);
  }
  if (a[j + 1] ≡ 0) break;
  if (a[j + 1] < 0) {
    fprintf(stderr, "Partition contains a negative element (%d)!\n", a[j + 1]);
    exit(−3);
  }
  if (a[j + 1] > maxn) {
    fprintf(stderr, "Partition element %d is too big (maxn=%d)!\n", a[j + 1], maxn);
    exit(−4);
  }
}
kk = j;
for (j = 2, n = a[1]; j ≤ kk; j++) n += a[j];
if (n > maxn) {
  fprintf(stderr, "Shape is too big (n=%d, maxn=%d)!\n", n, maxn);
  exit(−5);
}
```

This code is used in section 2.

5. This is exercise 7.2.1.4–6.

⟨ Compute the transposed shape λ^T 5 ⟩ ≡

```
for (k = a[1], j = 1; k; j++)
  while (k > a[j + 1]) b[k−−] = j;
```

This code is used in section 2.

6. Generating the standard tableaux. Here I use the Varol–Rotem algorithm to run through all the Young tableaux (Algorithm 7.2.1.2V).

All algorithms in this program are pretty much "brute force," with little attempt at optimization.

```
⟨ Find all the standard tableaux of the given shape 6 ⟩ ≡
  ⟨ Generate the order relation for the desired tableaux 7 ⟩;
v1: for (j = 0; j ≤ n; j++) p[j] = q[j] = j, prec[0][j] = 1;
v2: ⟨ Record the tableau represented by p and q 8 ⟩;
  k = n;
v3: j = q[k], l = p[j − 1];
  if (prec[l][k]) goto v5;
v4: p[j − 1] = k, p[j] = l, q[k] = j − 1, q[l] = j;
  goto v2;
v5: while (j < k) l = p[j + 1], p[j] = l, q[l] = j, j++;
  p[k] = q[k] = k;
  k−−;
  if (k) goto v3;
  ⟨ Assign index numbers to each tableau found 10 ⟩;
```

This code is used in section 2.

7.

```
⟨ Generate the order relation for the desired tableaux 7 ⟩ ≡
  for (j = jj = 0; j < kk; j++)
    for (k = 0; k < a[j + 1]; k++) {
      t[j][k] = ++jj;
      if (k > 0) prec[jj − 1][jj] = 1;
      if (j > 0) prec[t[j − 1][k]][jj] = 1;
    }
```

This code is used in section 6.

8. At this point we've found a standard tableau, whose entry in position $t[j][k]$ is $q[t[j][k]]$.

It is convenient to record a standard tableau as a permutation $w_1 \dots w_n$ of the multiset $\{a_1 \cdot 1, \dots, a_k \cdot k\}$, where the lth element of this permutation specifies the row occupied by the number l. Then a trie is used to keep track of all such permutations we've found.

```
⟨ Record the tableau represented by p and q 8 ⟩ ≡
  if (f ≡ maxf) {
    fprintf(stderr, "Too␣many␣standard␣tableaux␣exist␣(maxf=%d)!\n", maxf);
    exit(−6);
  }
  for (j = 0; j < kk; j++)
    for (k = 0; k < a[j + 1]; k++) w[q[t[j][k]]] = j + 1;
  for (j = 1, k = 0; j < n; j++, k = l) {
    l = trie[k][w[j]];
    if (l ≡ 0) l = trie[k][w[j]] = ++trienodes;
  }
  trie[k][w[n]] = 1;      /* mark a unique entry in the leif */
  f++;
```

This code is used in section 6.

9. ⟨ Global variables 3 ⟩ +≡

```
int prec[maxn + 1][maxn + 1];  /* prec[j][k] is nonzero if j ≺ k */
int w[maxn + 1];  /* codeword for a standard tableau */
int trie[maxf * maxn][maxn + 1];  /* trie memory, see Algorithm 6.3T */
int trienodes;  /* this many trie nodes have been allocated so far */
```

10. The standard tableaux are now given code numbers from 0 to $f - 1$. We walk through the trie in lexicographic order. (Yes, I could/should have done it recursively.)

⟨ Assign index numbers to each tableau found 10 ⟩ ≡

```
  l = 1, k = 0, j = 0;
newlev: w[l] = 1;
tryit: if (trie[k][w[l]]) {
    if (l ≡ n) {
      for ( ; l; l--) stand[j][l] = w[l];
      l = n;
      trie[k][w[l]] = j++;
      goto levdone;
    }
    q[l] = k, k = trie[k][w[l]], l++;
    goto newlev;
  }
tryagain: if (w[l] ≡ kk) goto levdone;
  w[l]++;
  goto tryit;
levdone: l--;
  if (l) {
    k = q[l];
    goto tryagain;
  }
  if (j ≠ f) {
    fprintf(stderr, "Oops, I goofed!\n");
    exit(-7);
  }
```

This code is used in section 6.

11. Finding admissible column perms. Now comes the heart of this program, the routine for solving the problem mentioned in the introduction.

Instead of producing a list of solutions, it sets $rep[j] = \pm 1$ for each standard tableau j achievable by column-then-row permutation, using the sign of the column permutation.

```
⟨ Subroutines 11 ⟩ ≡
  void findrep(void)      /* the input tableau is in t */
  {
    register int i, j, k, l, inv, sign;
    int row[maxn + 1], col[maxn + 1];      /* positions inside t */
    ⟨ Sort the columns of t 12 ⟩;
    for (j = 0; j < f; j++) rep[j] = 0;
    ⟨ Figure out where each element is, and set tt zero 13 ⟩;
    ⟨ Run through all solutions 14 ⟩;
  }
```

This code is used in section 2.

12. Insertion sort wins here, of course.

```
⟨ Sort the columns of t 12 ⟩ ≡
  inv = 0;
  for (k = 0; k < a[1]; k++) {
    for (j = 1; j < b[k + 1]; j++)
      if (t[j][k] < t[j − 1][k]) {
        for (i = j − 1, l = t[j][k]; ; i−−) {
          t[i + 1][k] = t[i][k];
          inv++;      /* inversions removed in this column */
          if (i ≡ 0 ∨ l > t[i − 1][k]) break;
        }
        t[i][k] = l;
      }
  }
```

This code is used in section 11.

13. ⟨ Figure out where each element is, and set tt zero 13 ⟩ ≡

```
  for (j = 0; j < kk; j++)
    for (k = 0; k < a[j + 1]; k++) {
      l = t[j][k];
      row[l] = j, col[l] = k;
      tt[j][k] = 0;
    }
```

This code is used in section 11.

14. Now we use a simple backtrack method to build a tableau tt from which row-sorting will be standard, by first placing 1 in tt, then 2, etc.

There always is at least one solution, because row sorting does not "mess up" column sorting. (See exercise 5.3.4–27.)

```
⟨ Run through all solutions 14 ⟩ ≡
  for (j = 1; j ≤ kk; j++) aa[j] = 0;
  aa[0] = maxn + 1;
  l = 1;
newlev: j = 1;
tryit: if (tt[j − 1][col[l]] ≡ 0 ∧ aa[j − 1] > aa[j]) {
    w[l] = j, tt[j − 1][col[l]] = l, aa[j]++;
    if (l ≡ n) ⟨ Use this solution and go to levdone 15 ⟩;
    l++;
    goto newlev;
  }
tryagain:
  if (++j ≤ b[col[l] + 1]) goto tryit;
levdone: l−−;
  if (l) {
    j = w[l], tt[j − 1][col[l]] = 0, aa[j]−−;
    goto tryagain;
  }
```

This code is used in section 11.

15.

```
⟨ Use this solution and go to levdone 15 ⟩ ≡
  {
    sign = inv;
    for (j = 1; j < kk; j++)
      for (k = 0; k < a[j + 1]; k++)
        for (l = 0; l < j; l++)
          if (tt[l][k] > tt[j][k]) sign++;
    for (k = 0, j = 1; j < n; j++) k = trie[k][w[j]];
    rep[trie[k][w[n]]] = (sign & 1 ? −1 : 1);
    l = n;
    j = w[l], tt[j − 1][col[l]] = 0, aa[j]−−;
    goto levdone;
  }
```

This code is used in section 14.

16. Finishing up. The theory to justify all these maneuverings can be found in many places; I wrote this program after reading Bruce Sagan's book *The Symmetric Group*, and Bruce based much of his treatment on G. D. James's monograph on representation theory [*Lecture Notes in Mathematics* **682** (1978)].

Those books use a more complicated "straightening rule" to compute representations and to prove important theorems. But once the theorems are proved, we can use them to justify the more direct approach taken here.

```
⟨ Compute f basis vectors for the representation 16 ⟩ ≡
  for (k = 0; k < f; k++) {
    for (j = 1; j ≤ kk; j++) aa[j] = 0;
    for (j = 1; j ≤ n; j++) l = stand[k][j], t[l − 1][aa[l]] = j, aa[l]++;
    findrep( );
    for (j = 0; j < f; j++) basis[k][j] = rep[j];
  }
```

This code is used in section 2.

17. ⟨ Read a permutation p (but **break** if there's no more good data) 17 ⟩ ≡

```
  for (j = 1; j ≤ n; j++)
    if (scanf("%d", &p[j]) ≠ 1) break;
  if (j ≤ n) break;
  for (j = 1; j ≤ n; j++) q[j] = 0;
  for (j = 1; j ≤ n; j++)
    if (p[j] ≤ 0 ∨ p[j] > n ∨ q[p[j]]) {
      fprintf(stderr, "Not␣a␣permutation␣of␣{1,...,%d}:", n);
      for (j = 1; j ≤ n; j++) fprintf(stderr, "␣%d", p[j]);
      fprintf(stderr, "!\n");
      exit(−8);
    } else q[p[j]] = j;
```

This code is used in section 2.

18. ⟨ Compute the representation of the jjth standard tableau, permuted by p 18 ⟩ ≡

```
  for (j = 1; j ≤ kk; j++) aa[j] = 0;
  for (j = 1; j ≤ n; j++) l = stand[jj][j], t[l − 1][aa[l]] = p[j], aa[l]++;
  findrep( );
```

This code is used in section 2.

19. ⟨ Reduce the representation to a linear combination of basis elements 19 ⟩ ≡

```
  for (j = 0; j < f; j++) {
    l = rep[j];
    if (l)
      for (k = j + 1; k < f; k++) rep[k] −= l ∗ basis[j][k];
  }
```

This code is used in section 2.

20. Index.

STRAIGHTEN