[This is a reprint, with slight amendments, of

Three Catalan Bijections

D. Knuth

Report No. 04, 2004/2005, spring
ISSN 1103-467X
ISRN IML-R- -04-04/05- -SE+spring

from Institut Mittag-Leffler of The Royal Swedish Academy of Sciences. It reports on work supported by the Swedish Research Council while the author was in residence at IML in Djursholm, Sweden, during January and February of 2005.]

Three Catalan Bijections

by Donald E. Knuth

Institut Mittag-Leffler and Stanford University

25 February 2005

This note contains three short programs that implement one-to-one correspondences between four kinds of combinatorial structures:

1) Ordered forests with n nodes and pruning order m;
2) Binary trees with n nodes and Strahler number m;
3) Nested strings (Dyck words) of length $2n$ and log-height m;
4) Kepler towers with n bricks and m walls.

In each case the number of structures of size n is the Catalan number $C_n = \binom{2n}{n}/(n+1)$, and — surprisingly — the bijections also preserve the parameter m.

Given a number $n > 1$, each program generates all C_n objects of one type, bijects them into objects of another type, verifies that the parameter m has not changed, and applies the inverse bijection to prove constructively that the correspondence is indeed one-to-one (at least for this value of n).

Program 1, called ZEILBERGER, converts between (1) and (2). Program 2, FRANÇON, converts between (2) and (3). And Program 3, VIENNOT, converts between (3) and (4). Incidentally, Kepler towers appear to be a completely new kind of object, recently invented by Xavier Viennot and introduced here for the first time. Simple bijections between (2) and (4), or between (1) and (4), are not yet known, although complex bijections could of course be obtained by composing those given here.

The first bijection was introduced by Doron Zeilberger in 1990, yet its computer implementation is not without interest. Although Zeilberger's algorithm was correct, his proof of correctness was not quite complete; Program 1 therefore removes any lingering doubts that may have existed. More significantly, the program demonstrates a strong property of Zeilberger's bijection that may not have been noticed before: Node x is the leftmost child of node y in the ordered forest if and only if node x is the left child of node y in the corresponding binary tree.

The second bijection was inspired by the work of Jean Françon in 1984, but it is organized here in a new way, based on a heap-like data structure. Therefore it appears to solve an open problem that he stated, namely to construct a "direct" parameter-preserving bijection between objects of types (2) and (3). Moreover, Program 2 has the interesting property that the bijection and its inverse both carry out their work in the same direction as they translate one object to another. By contrast, the inverse bijections in Programs 1 and 3 essentially cause time to run backward when they undo the effects of forward-running bijections.

Program 3 introduces a new bijection that was recently explained pictorially to the author by its creator, Xavier Viennot. The resulting computer program has turned out to be remarkably simple and fast.

All three programs have been written with the conventions of "literate programming," as embodied in the `CWEB` system developed by Silvio Levy and the author. This style of presentation features informal, human-oriented English descriptions alternating with formal, computer-oriented commands. The latter instructions are expressed in the C programming language; but mathematicians unfamiliar with C should still be able to get the gist of the ideas by reading the English commentary. (A detailed explanation of how to read `CWEB` programs — more than almost anybody needs to know — can be found in Chapter 4 of the author's book *The Stanford GraphBase*.)

Incidentally, these programs are independent of each other. They can be downloaded from the author's web site `http://www-cs-faculty.stanford.edu/~knuth/programs.html` and used without restriction.

Program 1

1. Introduction. This little program implements Zeilberger's bijection between n-node forests with pruning order s and n-node binary trees with Strahler number s. [Doron Zeilberger, "A bijection from ordered trees to binary trees that sends the pruning order to the Strahler number," *Discrete Mathematics* **82** (1990), 89–92.]

As Zeilberger says in his paper, "First, definitions!" Here I won't define forests (by which I always mean *ordered forests*, as in *The Art of Computer Programming*), nor need I define binary trees. But I should explain the concepts of pruning order and Strahler number, since they are less familiar.

A *filament* of a forest is a sequence of one or more nodes $x_1, \ldots, x_k$, where x_{j+1} is the only child of x_j for $1 \le j < k$, and where x_k is a leaf. Given a forest, we can *prune* it by removing all of its filaments. The number of times we must do this before reaching the empty forest is called the *pruning order* of the original forest.

The *Strahler number* of a binary tree is 0 if the binary tree is empty; otherwise it is $\max(s_l, s_r) + [s_l = s_r]$, when s_l and s_r are the Strahler numbers of the left and right binary subtrees of the root.

Zeilberger's bijection proves constructively the remarkable fact, discovered by Mireille Vauchaussade de Chaumont in her thesis (Bordeaux, 1985), that the pruning order and Strahler number have precisely the same distribution, when forests and binary trees are chosen uniformly at random.

Furthermore, as we shall see, his bijection has another significant property: When forests are represented in the natural way within a computer, as binary trees with left links to the leftmost child of a node and with right links to a node's right sibling, Zeilberger's transformation preserves all of the left links: Node x is the leftmost child of y in the original forest if and only if x is the left child of y in the binary tree that is produced by Zeilberger's procedure. In particular, the number of leaves in the forest equals the number of "left leaves" in the corresponding binary tree.

This programs runs through all forests with a given number of nodes, and computes the corresponding binary tree. It checks that the pruning order of the former equals the Strahler number of the latter; and then it applies the inverse bijection, thus verifying that the original forest can indeed be uniquely reconstructed.

2. Skarbek's algorithm (Algorithm 7.2.1.6B in *The Art of Computer Programming*, Volume 4) is used to run through all linked binary trees, thereby running through all forests in their natural representation.

```
#define n 17    /* nodes in the forest */

#include <stdio.h>
  int l[n + 2], r[n + 1];    /* leftmost child and right sibling */
  int ll[n + 1], rr[n + 1];    /* links of the binary tree */
  int lll[n + 1], rrr[n + 1];    /* links of the inverse forest */
  int q[n + 1], s[n + 1];    /* data needed by Zeilberger's algorithm */
  int serial;    /* total number of cases checked */
  int count[10];    /* individual counts by pruning order */
  ⟨ Subroutines 5 ⟩
  main()
  {
    register int j, k, y, p;
    printf("Checking␣all␣forests␣with␣%d␣nodes...\n", n);
    q[0] = 0, s[0] = 1000000;    /* see below */
    for (k = 1; k < n; k++) l[k] = k + 1, r[k] = 0;
    l[n] = r[n] = 0;    /* we start with the 1-filament forest */
    l[n + 1] = 1;    /* now Skarbek's algorithm is ready to go */
    while (1) {
      ⟨ Find the binary tree (ll, rr) corresponding to the forest (l, r) 6 ⟩;
      ⟨ Check the pruning order and Strahler number 13 ⟩;
      ⟨ Check the inverse bijection 14 ⟩;
      ⟨ Move to the next forest (l, r), or break 3 ⟩
    }
    for (p = 1; count[p]; p++) printf("Altogether␣%d␣cases␣with␣pruning␣order␣%d.\n", count[p], p);
  }
```

3. ⟨ Move to the next forest (l, r), or **break** 3 ⟩ ≡

```
  for (j = 1; ¬l[j]; j++) r[j] = 0, l[j] = j + 1;
  if (j > n) break;
  for (k = 0, y = l[j]; r[y]; k = y, y = r[y]) ;
  if (k > 0) r[k] = 0; else l[j] = 0;
  r[y] = r[j], r[j] = y;
```

This code is used in section 2.

4. The main algorithm. The nodes in the forest are represented by positive integers, according to their rank in preorder. For each node $x > 0$, we let $l[x]$ be its leftmost child (namely $x + 1$), if it has one, but $l[x] = 0$ when x is childless. Similarly, $r[x]$ is x's right sibling, or $r[x] = 0$ when x is rightmost in its family.

Zeilberger's method implicitly begins by determining the pruning order of the given forest and its principal subforests. We can carry this out by defining three numbers $p[x]$, $q[x]$, and $s[x]$ for each node x; here $p[x]$ is the pruning order of the subtree rooted at x, $q[x]$ is the maximum pruning order of x and all of its right siblings, and $s[x]$ is the number of nodes in which that maximum occurs. (Zeilberger called $s[l[x]]$ the number of "skewers" of x.)

Formally, we can compute $p[x]$, $q[x]$, and $s[x]$ via the following recursive definitions, if we set $q[0] = 0$ and $s[0] = \infty$:

$$p[x] = q[l[x]] + [s[l[x]] > 1];$$

$$q[x] = \max(p[x], q[r[x]]);$$

$$s[x] = \begin{cases} 1, & \text{if } p[x] > q[r[x]]; \\ s[r[x]] + 1, & \text{if } p[x] = q[r[x]]; \\ s[r[x]], & \text{if } p[x] < q[r[x]]. \end{cases}$$

5. The following recursive procedure computes $(p[x], q[x], s[x])$ at each node x of the forest. It turns out that $p[x]$ need not be stored in memory.

⟨ Subroutines 5 ⟩ ≡

```
void label(register int x)
{
  register int p, qr;
  if (l[x]) label(l[x]);
  if (r[x]) label(r[x]);
  p = q[l[x]] + (s[l[x]] > 1);
  qr = q[r[x]];
  if (p > qr) q[x] = p, s[x] = 1;
  else q[x] = qr, s[x] = s[r[x]] + (p ≡ qr);
}
```

See also sections 8, 12, and 15.

This code is used in section 2.

6. Zeilberger's bijection is recursively defined too, and we will implement it by writing another recursive procedure. But before we do so, let's examine how that procedure will be invoked in the program.

⟨ Find the binary tree (ll, rr) corresponding to the forest (l, r) 6 ⟩ ≡

```
for (k = 1; k ≤ n; k++) ll[k] = l[k], rr[k] = r[k];   /* clone the forest */
label(1);   /* compute all the q's and s's */
zeil(1);   /* transform the binary tree */
```

This code is used in section 2.

7. When subroutine $zeil(x)$ is called, x is a node of a forest, represented via left-child and right-sibling links ll and rr. The mission of $zeil(x)$ is to transform that forest in such a way that x will be a node of the final binary tree, while $ll[x]$ and $rr[x]$ will be the binary subtrees that are obtained by applying $zeil$ to two subforests.

Zeilberger's paper considered three cases, depending on the relative sizes of $q[x]$, $q[ll[x]]$, and $q[rr[x]]$. In Case 1, no change is needed; in Case 2, some of x's children are promoted to be siblings of x; in Case 3, some of x's children swap places with all of the right siblings.

8. ⟨ Subroutines 5 ⟩ +≡

```
void zeil(register int x)
{
  register int k, ql, qr, p, y, yy, z, zz, ss;
  k = q[x], ql = q[ll[x]], qr = q[rr[x]];
  if (ql ≡ k) ⟨ Do Zeilberger's Case 3 10 ⟩
  else if (qr < k − 1) ⟨ Do Zeilberger's Case 2 9 ⟩;
      /* otherwise we do Zeilberger's Case 1, which involves no action */
  if (ll[x]) zeil(ll[x]);
  if (rr[x]) zeil(rr[x]);
}
```

9. Case 2, the tricky case, detaches all but the first of x's "skewers."

When right links change, we must update the q and s numbers in each subnode of x before applying *zeil* to the new subforests. In particular, we must recompute all the q's and s's for the children of x that lie between the first and second skewer, because those nodes will no longer lie in the shadow of the second skewer.

This recomputation is a bit tricky because it must be done bottom-up, while our links go downward. An auxiliary recursive procedure could be introduced at this point, in order to achieve bottom-up behavior. But there's a more efficient (and more fun) way to do the job, namely to reverse the links temporarily as we descend the chain, and to reverse them again as we go back up.

⟨ Do Zeilberger's Case 2 9 ⟩ ≡

```
{
  for (y = ll[x], ss = s[y]; s[y] ≡ ss; yy = y, y = rr[yy]) s[y] = 1;
      /* at this point node yy is the first skewer of x */
  for (z = yy, ss−−; s[rr[y]] ≡ ss ∧ q[rr[y]] ≡ ql;
        yy = z, z = y, y = rr[z], rr[z] = yy) ;    /* reverse links */
      /* now node y is the second skewer of x */
      /* its left sibling, z, will become the last child of x */
  if (z ≡ yy) rr[z] = 0;    /* easy case: the skewers were adjacent */
  else for (zz = 0; rr[z] ≠ zz; yy = zz, zz = z, z = rr[zz], rr[zz] = yy) {
      p = q[ll[z]] + (s[ll[z]] > 1);    /* we will recompute q[z] and s[z] */
      if (p > q[zz]) q[z] = p, s[z] = 1;
      else q[z] = q[zz], s[z] = s[zz] + (p ≡ q[zz]);
    }
  for (zz = x, z = rr[zz]; z; zz = z, z = rr[zz]) q[z] = ql, s[z] = ss;
  rr[zz] = y;    /* y and its right siblings become x's final siblings */
}
```

This code is used in section 8.

10. In Case 3, x has only one skewer. But the transformation in this case demotes siblings to children, where they might become additional skewers. It also might promote some children to siblings.

⟨ Do Zeilberger's Case 3 10 ⟩ ≡

```
{
  if (qr ≡ k) ss = s[rr[x]] + 1; else ss = 1;
  for (y = ll[x]; q[y] ≡ k; yy = y, y = rr[yy]) s[y] = ss;
  rr[yy] = rr[x], rr[x] = y;
}
```

This code is used in section 8.

11. Checking the Strahler number. If our implementation of Zeilberger's transformation is correct, it will have set $q[x]$ to the Strahler number of the binary subtree rooted at x with respect to the *ll* and *rr* links, for every node x.

Therefore we want to check this condition. And we might as well do the checking by brute force, so that the evidence is convincing.

12. ⟨ Subroutines 5 ⟩ +≡
```
int strahler(register int x)
{
  register int sl, sr, s;
  if (ll[x]) sl = strahler(ll[x]);
  else sl = 0;
  if (rr[x]) sr = strahler(rr[x]);
  else sr = 0;
  s = (sl > sr ? sl : sl < sr ? sr : sl + 1);
  if (q[x] != s) fprintf(stderr, "I goofed at binary tree node %d, case %d.\n", x, serial);
  return s;
}
```

13. ⟨ Check the pruning order and Strahler number 13 ⟩ ≡
```
count[strahler(1)]++;
serial++;
```
This code is used in section 2.

14. The inverse algorithm. The evidence of correctness is mounting. But our argument in favor of Zeilberger's algorithm is still not compelling, because there are lots of ways to convert a forest with pruning order s into a binary tree with Strahler number s. For example, we need only compute the pruning order, then choose our favorite binary tree that has the desired Strahler number.

A bijection must do more than this: It must not destroy information. We must be able to go back from each binary tree to the original forest that produced it. Thus, our implementation of Zeilberger's procedure is incomplete until we have also implemented its inverse, *unzeil*.

Our *unzeil* algorithm will recreate the forest in new arrays *lll* and *rrr*, just to emphasize that no cheating is going on.

⟨ Check the inverse bijection 14 ⟩ ≡

```
for (k = 1; k ≤ n; k++) lll[k] = ll[k], rrr[k] = rr[k], q[k] = s[k] = 0;    /* clone the binary tree */
unzeil(1);    /* attempt to recreate the original forest */
for (k = 1; k ≤ n; k++)
  if (lll[k] ≠ l[k] ∨ rrr[k] ≠ r[k]) fprintf(stderr, "Rejection␣at␣node␣%d␣of␣case␣%d!\n", k, serial);
```

This code is used in section 2.

15. The *unzeil* procedure also computes the q and s values at each node of the reconstructed forest.

⟨ Subroutines 5 ⟩ +≡

```
void unzeil(register int x)
{
  register int ql, qr, p, y, yy, z, zz, ss;
  if (rrr[x]) unzeil(rrr[x]);
  if (lll[x]) unzeil(lll[x]);
  ql = q[lll[x]], qr = q[rrr[x]];
  if (ql > qr) ⟨ Undo Zeilberger's Case 3 16 ⟩
  else if (ql ≡ qr ∧ s[lll[x]] ≡ 1) ⟨ Undo Zeilberger's Case 2 17 ⟩;
      /* otherwise we undo Zeilberger's Case 1, which involves no action */
  p = ql + (s[lll[x]] > 1);
  if (p > qr) q[x] = p, s[x] = 1;
  else q[x] = qr, s[x] = s[rrr[x]] + (p ≡ qr);
}
```

16. ⟨ Undo Zeilberger's Case 3 16 ⟩ ≡

```
{
  if (s[lll[x]] > 1) {
    for (y = lll[x]; s[rrr[y]] ≡ s[y]; y = rrr[y]) s[y] = 1;
    s[y] = 1;
  } else for (y = lll[x]; q[rrr[y]] ≡ q[y]; y = rrr[y]) ;
  yy = rrr[y], rrr[y] = rrr[x], rrr[x] = yy;    /* swap siblings with children */
  qr = q[yy];    /* the subsequent program assumes that qr = q[rrr[x]] */
}
```

This code is used in section 15.

17. We have saved the other tricky case for last: Again we do a double-reversal in order to recompute any q's and s's that have been obliterated. In this case the recomputation affects the near siblings of x.

⟨ Undo Zeilberger's Case 2 17 ⟩ ≡

```
{
  for (z = rrr[x], zz = x; s[rrr[z]] ≡ s[z] ∧ q[rrr[z]] ≡ q[z];
        yy = zz, zz = z, z = rrr[zz], rrr[zz] = yy) ;
  if (zz ≡ x) yy = rrr[x] = 0;
  else for (yy = 0; zz ≠ x; y = zz, zz = rrr[y], rrr[y] = yy, yy = y) {
      p = q[lll[zz]] + (s[lll[zz]] > 1);     /* we will recompute q[zz] and s[zz] */
      if (p > q[yy]) q[zz] = p, s[zz] = 1;
      else q[zz] = q[yy], s[zz] = s[yy] = (p ≡ q[yy]);
    }     /* x's former siblings are now undone */
  qr = q[yy], ss = s[z] + 1;     /* at this point yy = rrr[x] */
  for (yy = lll[x], y = rrr[yy]; y; yy = y, y = rrr[yy]) {
    s[yy] = ss;
    if (q[yy] ≡ ql) ss--;
  }
  s[yy] = ss, rrr[yy] = z;     /* unpromote x's former children */
}
```

This code is used in section 15.

Program 2

1. Introduction. This short program implements a Françon-inspired bijection between binary trees with Strahler number s and nested strings with height h, where $2^s - 1 \le h < 2^{s+1} - 1$. But it uses a direct method that is complementary to his approach. [Reference: Jean Françon, "Sur le nombre de registres nécessaires a l'évaluation d'une expression arithmétique," *R.A.I.R.O. Informatique théorique* **18** (1984), 355–364.]

```
#define n 17   /* nodes in the tree */
#define nn (n + n)
#include <stdio.h>
  int d[nn + 1];   /* the path, a sequence of ±1s */
  int l[n + 1], r[n + 1];   /* tree links */
  int h[nn + 1], q[n + 1], qm[n + 1];   /* heap and queue structures for decision-making */
  int serial;   /* total number of cases checked */
  int count[10];   /* individual counts by Strahler number */
  ⟨ Subroutines 5 ⟩
  main( )
  {
    register int i, j, k, jj, kk, m, p, s;
    printf("Checking␣binary␣trees␣with␣%d␣nodes...\n", n);
    ⟨ Set up the first nested string, d 2 ⟩;
    while (1) {
      ⟨ Find the tree corresponding to d 7 ⟩;
      ⟨ Check the Strahler number 4 ⟩;
      ⟨ Check the inverse bijection 9 ⟩;
      ⟨ Move to the next nested string, or goto done 3 ⟩;
    }
  done:
    for (s = 1; count[s]; s++)
      printf("Altogether␣%d␣cases␣with␣Strahler␣number␣%d.\n", count[s], s);
  }
```

2. Nested strings (aka Dyck words) are conveniently generated by Algorithm 7.2.1.6P of *The Art of Computer Programming*.

```
⟨ Set up the first nested string, d 2 ⟩ ≡
  for (k = 0; k < nn; k += 2) d[k] = +1, d[k + 1] = −1;
  d[nn] = −1, i = nn − 2;
```

This code is used in section 1.

3. At this point, variable i is the position of the rightmost '+1' in d.

```
⟨ Move to the next nested string, or goto done 3 ⟩ ≡
  d[i] = −1;
  if (d[i − 1] < 0) d[i − 1] = 1, i−−;
  else {
    for (j = i − 1, k = nn − 2; d[j] > 0; j−−, k −= 2) {
      d[j] = −1, d[k] = +1;
      if (j ≡ 0) goto done;
    }
    d[j] = +1, i = nn − 2;
  }
```

This code is used in section 1.

4. ⟨ Check the Strahler number 4 ⟩ ≡

```
for (s = j = k = 1; k < nn - 1; j += d[k], k++)
  if (j >= ((1 << s) - 1)) s++;
s--;   /* now s is the Strahler number */
count[s]++, serial++;
if (strahler(1) != s) {
  fprintf(stderr, "I goofed on case %d.\n", serial);
}
```

This code is used in section 1.

5. ⟨ Subroutines 5 ⟩ ≡

```
int strahler(int x)
{
  register int sl, sr;

  if (l[x]) sl = strahler(l[x]);
  else sl = 0;
  if (r[x]) sr = strahler(r[x]);
  else sr = 0;
  return (sl > sr ? sl : sl < sr ? sr : sl + 1);
}
```

This code is used in section 1.

6. The main algorithm. A large family of bijections between nested strings and binary trees was described by Proskurowski in *JACM* **27** (1980), page 1: We build a binary tree by choosing, at each step, some node and some yet-unset link field in that node; then we look at the next element $d[p]$ of the nested string. The link is set to a new node if $d[p] > 0$, and to null if $d[p] < 0$. The bijection implemented here is of that type.

To decide what link should be constructed next, we use a heap-like data structure $h[1]$, $h[2]$, $\ldots$, in which cell k is the parent of cells $2k$ and $2k+1$. The cell elements are pointers to nodes in the tree being built, and the nodes recorded in the heap can be embedded as a subtree of that tree. (In other words, if $h[k]$ and $h[\lfloor k/2 \rfloor]$ are both nonzero, they point to nodes of the tree in which the first is a descendant of the second. It might be helpful to imagine a set of pebbles on the tree, with the heap cells recording the positions of those pebbles.) When $h[2k] = 0$, meaning that heap cell $2k$ is empty, we also have $h[2k+1] = 0$. The basic idea of the algorithm is to attempt to fill the first empty cell k in the heap, by setting the links of the tree node pointed to by $h[k/2]$.

The number of elements in the heap is always the partial sum $d[0] + \cdots + d[p]$. If this number is $2^t - 1$ or more, the Strahler number of the binary tree is at least t. Conversely, if the Strahler number is s, one can show without difficulty that the partial sum will indeed reach the value $2^s - 1$ at some point, with the heap at that time containing the "topmost" complete subtree of size $2^s - 1$ embedded in the tree.

For validity of this algorithm, we don't really need to choose the first hole in the heap. Any rule for choosing k would work, provided only that (a) k is even; (b) $h[k/2] \neq 0$; and (c) $k \geq 2^t$ implies $d[0] + \cdots + d[p] \geq 2^t - 1$. Thus there are many possible bijections, some of which are presumably easier to analyze than others.

7. Variable m represents the number of nodes in the tree; variable p is our position in the nested string; and variable k is a lower bound on the location of the least hole in the heap.

```
⟨ Find the tree corresponding to d 7 ⟩ ≡
  h[1] = m = 1, k = 2, p = 0;
  while (1) {
    while (h[k]) k += 2;    /* find the smallest hole */
    kk = h[k >> 1];    /* kk is the node pointed to by k's parent */
    if (d[++p] > 0) h[k] = l[kk] = ++m; else l[kk] = 0;
    if (d[++p] > 0) h[k + 1] = r[kk] = ++m; else r[kk] = 0;
    if (h[k]) {
      if (h[k + 1]) continue;
      kk = k;
    } else if (h[k + 1]) kk = k + 1;
    else {
      h[k >> 1] = 0, kk = (k >> 1) ⊕ 1, k = kk & −2;
      if (k ≡ 0) break;    /* we're done when the heap is empty */
    }
    ⟨ Move the subheap rooted at kk up one level 8 ⟩;
  }
```

This code is used in section 1.

8. Let the binary representation of kk be $(b_t \ldots b_0)_2$. We want to set $h[(b_t \ldots b_1\alpha)_2] \leftarrow h[(b_t \ldots b_0\alpha)_2]$ for all binary strings α.

$\langle$ Move the subheap rooted at kk up one level 8 $\rangle \equiv$

```
j = 0, jj = 1, q[0] = kk, qm[0] = 1;
while (j < jj) {
    kk = q[j];
    h[((kk >> 1) & -qm[j]) + (kk & (qm[j] - 1))] = h[kk];
    if (h[kk + kk]) q[jj] = kk + kk, q[jj + 1] = kk + kk + 1, qm[jj] = qm[jj + 1] = qm[j] << 1, jj += 2;
    else h[kk] = 0;
    j++;
}
```

This code is used in sections 7 and 9.

9. The inverse algorithm. To reverse the process, we simply look at the tree and build the nested string, instead of vice versa. The same heap-oriented logic applies.

#define $\mathit{check}(s)$
  { **if** $(d[{+}{+}p] \neq s)$ $\mathit{fprintf}(\mathit{stderr},$ `"Rejection␣at␣position␣%d␣of␣case␣%d!\n"`$, p, \mathit{serial});$ }

⟨ Check the inverse bijection 9 ⟩ ≡

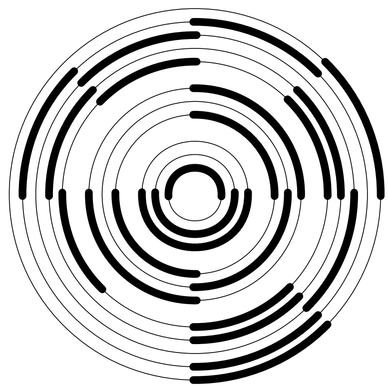
```
h[1] = 1, k = 2, p = 0;
while (1) {
  while (h[k]) k += 2;    /* find the smallest hole */
  kk = h[k ≫ 1];    /* kk is the node pointed to by k's parent */
  if (l[kk]) {
    h[k] = l[kk]; check(+1);
  } else check(−1);
  if (r[kk]) {
    h[k + 1] = r[kk]; check(+1);
  } else check(−1);
  if (h[k]) {
    if (h[k + 1]) continue;
    kk = k;
  } else if (h[k + 1]) kk = k + 1;
  else {
    h[k ≫ 1] = 0, kk = (k ≫ 1) ⊕ 1, k = kk & −2;
    if (k ≡ 0) break;    /* we're done when the heap is empty */
  }
  ⟨ Move the subheap rooted at kk up one level 8 ⟩;
}
```

This code is used in section 1.

Program 3

1. Introduction. This short program implements a Viennot-inspired bijection between Kepler towers with w walls and nested strings with height h, where $2^w - 1 \le h < 2^{w+1} - 1$.

What is a Kepler tower? Good question. It is a new kind of combinatorial object, invented by Xavier Viennot in February 2005. For example,



depicts a Kepler tower with 3 walls containing 22 bricks. This illustration is two-dimensional, but of course a tower has three dimensions; we are viewing the tower from above. Every wall of a Kepler tower consists of one or more rings, where each ring of the kth wall is divided into 2^k segments of equal length. Each brick is slightly longer than the length of one segment. Viennot gave the name Kepler tower to such a configuration because it somehow suggests Kepler's model of the solar system, with the sun in the center and the planets surrounding it in circumscribed shells.

Brick positions in the rings of the kth wall are identified by a sequence of segment numbers p_1, ..., p_t, where $1 \le p_1 < \cdots < p_t \le 2^k$. For example, the Kepler tower above is specified by the following segment-number sequences:

$$1;\ 2;\ 2;\quad 1,3;\ 4;\ 1,3;\quad 1,3,5,7;\ 1,4,7;\ 3,8;\ 2,4,7;\ 1,7.$$

(In the diagram above, segment 1 of every ring begins due east of the center, and we reach segments 2, 3, ... by proceeding counterclockwise from there.) These sequences must satisfy three constraints:

i) The positions in the first (bottom-most) ring in the kth wall must be 1, 3, ..., $2^k - 1$, for $1 \le k \le s$.

ii) Bricks cannot occupy adjacent segments of a ring. In other words, consecutive positions (p_j, p_{j+1}) in each ring must differ by at least 2, and the case $(p_t, p_1) = (2^k, 1)$ is also forbidden.

iii) Bricks in each non-bottom ring must be in contact with bricks in the ring below. In other words, whenever p_j is the number of an occupied segment in a ring of the kth wall, not at the base of that wall, the ring below it must contain at least one brick in segment number $p_j - 1$, p_j, or $p_j + 1$ (modulo 2^k).

(Note to construction workers and LEGO fans: The walls also contain little struts, not shown in the diagram, which keep the bricks of each ring from tipping over.)

An auxiliary program, BACK-KEPLER-TOWERS, generates all Kepler towers by "brute force," directly from this definition. One can use it to verify experimentally that the number of Kepler towers with n bricks is exactly the Catalan number C_n, if n isn't too large.

2. And what is a nested string? A nested string (aka Dyck word) of order n is a sequence $d_0, d_1, \ldots, d_{2n-1}$ of ±1s whose partial sums $y_k = d_0 + \cdots + d_k$ are nonnegative, and whose overall sum y_{2n} is zero. Its height is $\max_{0\le k<2n} y_k$. The bijection implemented in this program associates the example tower above with the nested string having

as its graph of partial sums; in this case the height is 11.

```
#define n 17      /* bricks in the tower */
#define nn (n + n)      /* elements in the nested string */
#include <stdio.h>
  int d[nn + 1];      /* the path, a sequence of ±1s */
  int x[nn + 1];      /* partial sums of the d's */
  char ring[n][n + 3], ringcount[n];      /* occupied segments */
  int wall[n];      /* wall boundaries in the ring array */
  int serial;      /* total number of cases checked */
  int count[10];      /* individual counts by number of walls */
  main()
  {
    register int i, j, k, m, p, w, ww, y, mode;
    printf("Checking␣Kepler␣towers␣with␣%d␣bricks...\n", n);
    ⟨Set up the first nested string, d 3⟩;
    while (1) {
      ⟨Find the tower corresponding to d 7⟩;
      ⟨Check the number of walls 5⟩;
      ⟨Check the inverse bijection 10⟩;
      ⟨Move to the next nested string, or goto done 4⟩;
    }
  done:
    for (w = 1; count[w]; w++)
      printf("Altogether␣%d␣cases␣with␣%d␣wall%s.\n", count[w], w, w > 1 ? "s" : "");
  }
```

3. Nested strings are conveniently generated by Algorithm 7.2.1.6P of *The Art of Computer Programming*.

⟨ Set up the first nested string, d 3 ⟩ ≡

```
for (k = 0; k < nn; k += 2) d[k] = +1, d[k + 1] = −1;
d[nn] = −1, i = nn − 2;
```

This code is used in section 2.

4. At this point, variable i is the position of the rightmost '+1' in d.

⟨ Move to the next nested string, or **goto** *done* 4 ⟩ ≡

```
d[i] = −1;
if (d[i − 1] < 0) d[i − 1] = 1, i−−;
else {
  for (j = i − 1, k = nn − 2; d[j] > 0; j−−, k −= 2) {
    d[j] = −1, d[k] = +1;
    if (j ≡ 0) goto done;
  }
  d[j] = +1, i = nn − 2;
}
```

This code is used in section 2.

5. ⟨ Check the number of walls 5 ⟩ ≡

```
for (m = j = k = 1; k < nn − 1; j += d[k], k++)
  if (j ≥ ((1 ≪ m) − 1)) m++;
m−−;     /* now there should be m walls */
count[m]++, serial++;
if (w ≠ m) {
  fprintf(stderr, "I␣goofed␣on␣case␣%d.\n", serial);
}
```

This code is used in section 2.

6. The main algorithm. Given a nested string of order n, we append $d_{2n} = -1$ so that the total sum is -1. Then we read it sequentially and begin to construct the kth wall of the corresponding Kepler tower at the moment the partial sum $d_0 + \cdots + d_p$ first reaches the value $2^k - 1$. (Thus, for example, we build the first wall immediately, unless $n = 0$, because d_0 is always $+1$ when $n > 0$.)

The main idea is to associate every r-segment wall with a sequence of ± 1s whose partial sums remain strictly less than r in absolute value, except that the total sum is $-r$. Let's call this an r-path. For example, a one-wall Kepler tower corresponds to a sequence d_0, d_1, ..., d_{2n} whose partial sums remain nonnegative until the last step, and never exceed 2. Removing the first element, d_0, leaves a 2-path, because its partial sums are always 0, 1, or -1, until finally reaching $d_1 + \cdots + d_{2n} = -2$.

The kth wall of a larger tower will correspond to a 2^k-path in a similar fashion. For example, the outer wall of a 3-wall tower comes from a sequence d_{p+1}, ..., d_{2n} whose partial sums lie between -7 and $+7$, except that the total sum is -8; here p denotes the smallest subscript such that $d_0 + \cdots + d_p = 7$.

There's a slight problem, however, because the inner walls don't behave in the same way; they give a "dual" r-path (the negative of a true r-path), in which the total sum is $+r$ instead of $-r$. Furthermore, our rule that associates r-paths with r-segment walls doesn't obey rule (i) of Keplerian walls: Our rule describes only the bricks *above* the bottom ring; it produces one brick for each $+1$ in the r-path, so it might not produce any bricks at all.

The solution is to associate both an ordinary wall and a dual wall with any r-path or dual r-path, where the ordinary wall has a brick for each $+1$ and the dual wall has a brick for each -1. These walls won't satisfy rule (i), the bottom-ring constraint; but when we combine them properly, everything fits together nicely so that perfect Keplerian walls are indeed produced.

The reason this plan succeeds can best be understood by considering what happens when a nested string $(d_0, d_1, \ldots, d_{2n})$ corresponds to, say, a 3-wall Kepler tower. Such a path begins with $d_0 = +1$; then comes a dual 2-path, ending at d_{p_1}, containing say n_1 positive elements and $n_1 - 2$ negative elements, so that $p_1 = 2n_1 - 2$. A dual 4-path begins at d_{p_1+1} and ends at d_{p_2}, containing n_2 occurrences of $+1$ and $n_2 - 4$ occurrences of -1, so that $p_2 = p_1 + 2n_2 - 4$. Finally there is an ordinary 8-path containing n_3 positives and $n_3 + 8$ negatives, so that $2n = p_2 + 2n_3 + 8 = 2n_1 + 2n_2 + 2n_3 + 2$. We obtain the desired Kepler tower by putting one brick on the bottom ring and placing $n_1 - 2$ bricks above them, using the dual wall from the dual 2-path. We also put two bricks on the bottom ring of the second wall and place $n_2 - 4$ bricks above them, using the dual wall from the dual 4-path. And finally we put four bricks on the bottom ring of the outer wall, surmounted by the n_3 bricks that represent the ordinary wall of the ordinary 8-path. The total number of bricks is $1 + n_1 + n_2 + n_3 = n$, as desired.

In summary, the problem is solved if we can find a good way to produce r-segment walls from r-paths. And indeed, there is a simple bijection: When the partial sum changes from 0 to 1, go into "downward mode" in which a brick drops into segment s when the partial sum decreases from s to $s - 1$. When the partial sum changes from 0 to -1, go into "upward mode" in which a brick drops into segment s when the partial sum increases from $s - r - 1$ to $s - r$. In either case, bricks drop into the uppermost ring for which they currently have support from below.

For example, let's consider the case $r = 3$. (Kepler towers use only cases where r is a power of 2, but the bijection in the previous paragraph works fine for any value of $r \geq 2$.) The 3-path

$$+1,\ +1,\ -1,\ +1,\ -1,\ -1,\ -1,\ +1,\ -1,\ +1,\ +1,\ -1,\ -1,\ -1,\ +1,\ -1,\ -1$$

with partial sums

$$+1,\ +2,\ +1,\ +2,\ +1,\quad 0,\ -1,\quad 0,\ -1,\quad 0,\ +1,\quad 0,\ -1,\ -2,\ -1,\ -2,\ -3$$

goes into downward mode, drops bricks in segments 2, 2, 1, then goes into upward mode, drops a brick in segment 3, enters upward mode again and drops another 3, then resumes downward mode and drops a brick into 1, and finishes with upward mode and a brick into 2. (When $r = 3$ each brick begins a new ring when it is dropped, because at most $\lfloor r/2 \rfloor$ bricks fit on a single ring.) We can reverse the process and reconstruct the original sequence by removing bricks from top to bottom, as described below.

7. This program represents an r-segment ring as an array of $r+2$ bytes, numbered 0 to $r+1$, with byte k equal to 1 or 0 according as a brick occupies segment k or not. Byte 0 is a duplicate of byte r, and byte $r+1$ is a duplicate of byte 1, so that we can easily test whether a brick will fit in a given segment of a given ring.

The current state of the tower appears in $ring[0]$, $ring[1]$, ..., $ring[m]$, where each $ring[j]$ is an array of bytes as just mentioned. The number of bricks in $ring[j]$ is maintained in $ringcount[j]$. Variable w is the current number of walls; and the kth wall consists of $ring[j]$ for $wall[k-1] \le j < wall[k]$, for $1 \le k \le w$. If we have most recently looked at $d[p]$, variable y is the partial sum $d[p'+1] + \cdots + d[p]$ of the current 2^w-path, where p' denotes the position where the 2^{w-1}-path ended.

We shall assume that all elements of $ring$ are identically zero when this algorithm begins.

```
⟨Find the tower corresponding to d 7⟩ ≡
  w = 1, ww = 2, m = 0;      /* ww = 2^w */
  ring[0][1] = ring[0][3] = 1;
  for (y = p = 0; y ≠ −ww; ) {
    if (y ≡ 0) mode = −d[++p], y −= mode;
    else if (y ≡ ww) ⟨Begin a new wall 8⟩
    else {
      y += d[++p];
      if (d[p] ≡ mode) ⟨Place a brick 9⟩;
    }
  }
  wall[w] = m + 1;
```

This code is used in section 2.

8.
```
⟨Begin a new wall 8⟩ ≡
  {
    wall[w++] = ++m;
    ww += ww;
    for (k = 0; k ≤ ww; k += 2) ring[m][k + 1] = 1;
    y = 0;
  }
```

This code is used in section 7.

9.
```
⟨Place a brick 9⟩ ≡
  {
    k = y + (mode < 0 ? 1 : ww);      /* we'll drop a brick into segment k */
    for (j = m; ring[j][k − 1] ≡ 0 ∧ ring[j][k] ≡ 0 ∧ ring[j][k + 1] ≡ 0; j−−) ;
    if (j ≡ m) m++;      /* enter a new ring, initially empty */
    ring[j + 1][k] = 1;
    if (k ≡ 1) ring[j + 1][ww + 1] = 1;
    else if (k ≡ ww) ring[j + 1][0] = 1;
    ringcount[j + 1]++;
  }
```

This code is used in section 7.

10. The inverse algorithm. Going backward is a matter of removing bricks in the reverse order, reconstructing the nested string that must have produced them. At the end of this process, the *ring* array will once again be identically zero.

#define $\mathit{check}(s)$
$\{\ y \mathrel{-}= d[{-}{-}p];$
if $(d[p] \neq s)$ $\mathit{fprintf}(\mathit{stderr},$ `"Rejection␣at␣position␣%d,␣case␣%d!\n"`$, p, \mathit{serial});\ \}$

⟨ Check the inverse bijection 10 ⟩ ≡
$m = \mathit{wall}[w] - 1, \mathit{mode} = +1;$
for $(y = -\mathit{ww} + 1, p = \mathit{nn};\ p;\ )\ \{$
if $(y \equiv 1 - \mathit{ww} \lor y \equiv \mathit{ww} - 1)$ $\mathit{check}(-\mathit{mode})$
else ⟨ Remove a brick if it's free and ready, or $\mathit{check}(-\mathit{mode})$ 11 ⟩;
}

This code is used in section 2.

11. ⟨ Remove a brick if it's free and ready, or $\mathit{check}(-\mathit{mode})$ 11 ⟩ ≡
{
look: $k = y + (\mathit{mode} < 0\ ?\ 1 : \mathit{ww});$ /* we'll look for a brick in segment k */
for $(j = m;\ \mathit{ring}[j][k] \equiv 0;\ j{-}{-})$
if $(\mathit{ring}[j][k-1] \lor \mathit{ring}[j][k+1])$ **goto** *notfound*;
if $(j \equiv \mathit{wall}[w-1])$ **goto** *notfound*;
$\mathit{ring}[j][k] = 0;$ /* we found it! out it goes */
if $(k \equiv 1)$ $\mathit{ring}[j][\mathit{ww}+1] = 0;$
else if $(k \equiv \mathit{ww})$ $\mathit{ring}[j][0] = 0;$
$\mathit{ringcount}[j]{-}{-};$
if $(\mathit{ringcount}[j] \equiv 0)$ $m{-}{-};$
$\mathit{check}(\mathit{mode});$ **continue**;
notfound: **if** $(y \equiv 0)$ {
if $(m \equiv \mathit{wall}[w-1])$ ⟨ Remove a wall's base 12 ⟩
else ⟨ Change the mode and **goto** *look* 13 ⟩;
}
$\mathit{check}(-\mathit{mode});$
}

This code is used in section 10.

12. ⟨ Remove a wall's base 12 ⟩ ≡
{
for $(k = 0;\ k \le \mathit{ww};\ k \mathrel{+}= 2)$ $\mathit{ring}[m][k+1] = 0;$
$m{-}{-}, \mathit{ww} \gg= 1, w{-}{-};$
$y = \mathit{ww}, \mathit{mode} = -1;$
}

This code is used in section 11.

13. If $y = 0$ and $\mathit{mode} > 0$, we looked for a brick in segment *ww* and didn't find it. But at least one brick remains in the current wall. Therefore, by the nature of the algorithm, a brick must be free in segment 1. Similarly, if $y = 0$ and $\mathit{mode} < 0$, there must be a free brick in segment *ww* at this time. (Think about it.)

⟨ Change the mode and **goto** *look* 13 ⟩ ≡
{
$\mathit{mode} = -\mathit{mode};$ **goto** *look*;
}

This code is used in section 11.