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

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!" 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, ..., 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 ($ql \equiv k$) ⟨ 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 ($y = ll[x], ss = s[y]$; $s[y] \equiv ss$; $yy = y, y = rr[yy]$) $s[y] = 1$;
      /* at this point node yy is the first skewer of x */
  for ($z = yy, ss--$; $s[rr[y]] \equiv ss \land q[rr[y]] \equiv ql$;
         $yy = z, z = y, y = rr[z], rr[z] = yy$) ;     /* reverse links */
      /* now node y is the second skewer of x */
      /* its left sibling, z, will become the last child of x */
  if ($z \equiv yy$) $rr[z] = 0$;     /* easy case: the skewers were adjacent */
  else for ($zz = 0$; $rr[z] \neq zz$; $yy = zz, zz = z, z = rr[zz], rr[zz] = yy$) {
      $p = q[ll[z]] + (s[ll[z]] > 1)$;     /* we will recompute q[z] and s[z] */
      if ($p > q[zz]$) $q[z] = p, s[z] = 1$;
      else $q[z] = q[zz], s[z] = s[zz] + (p \equiv q[zz])$;
    }
  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 ($qr \equiv k$) $ss = s[rr[x]] + 1$; else $ss = 1$;
  for ($y = ll[x]$; $q[y] \equiv k$; $yy = y, y = rr[yy]$) $s[y] = ss$;
  $rr[yy] = rr[x], rr[x] = y$;
}
```

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 ($z = rrr[x], zz = x;$ $s[rrr[z]] \equiv s[z] \land q[rrr[z]] \equiv q[z];$
          $yy = zz, zz = z, z = rrr[zz], rrr[zz] = yy$) ;
    if ($zz \equiv x$) $yy = rrr[x] = 0;$
    else for ($yy = 0;$ $zz \neq x;$ $y = zz, zz = rrr[y], rrr[y] = yy, yy = y$) {
        $p = q[lll[zz]] + (s[lll[zz]] > 1);$      /* we will recompute $q[zz]$ and $s[zz]$ */
        if ($p > q[yy]$) $q[zz] = p, s[zz] = 1;$
        else $q[zz] = q[yy], s[zz] = s[yy] = (p \equiv q[yy]);$
      }      /* $x$'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 ($q[yy] \equiv ql$) $ss--;$
    }
    $s[yy] = ss, rrr[yy] = z;$      /* unpromote $x$'s former children */
  }
```

This code is used in section 15.

18. Index.

ZEILBERGER