**1. Intro.** This (hastily written) program computes the Baxter permutation that corresponds to a given twintree. See exercises MPR–135 and 7.2.2.1–372 in Volume 4B of *The Art of Computer Programming* for an introduction to the relevant concepts and terminology.

According to exercise 7.2.2.1–372, a twintree is a data structure characterized by the following interesting properties: (i) There are $n$ nodes, each of which has four fields called $l_0$, $r_0$, $l_1$, $r_1$. (ii) The $l_0$ and $r_0$ fields are the left and right links of a binary tree $T_0$ that's rooted at node $t_0$. (iii) The $l_1$ and $r_1$ fields are the left and right links of a binary tree $T_1$ that's rooted at node $t_1$. (iv) The symmetric order of both trees (also called "inorder") is $1\,2\ldots n$. (v) For $1 \le k < n$, $r_0[k]$ is null if and only if $r_1[k]$ is nonnull.

Condition (iv) is equivalent to saying that, for $1 \le k \le n$, $l_0[k]$ and $l_1[k]$ are either null or less than $k$; $r_0[k]$ and $r_1[k]$ are either null or greater than $k$.

Condition (v) might seem surprising at first, possibly even weird. But it's not hard to see that a twintree structure can indeed be obtained from any permutation $P = p_1p_2\ldots p_n$ of $\{1, 2, \ldots, n\}$, as follows: Create $T_0$ by using the classic binary tree insertion procedure, Algorithm 6.2.2T, to insert $p_1$, then $p_2$, ..., then $p_n$, into an initially empty binary tree. Create $T_1$ by using that same procedure to insert $p_n$, then $p_{n-1}$, ..., then $p_1$. The roots of those trees, $t_0$ and $t_1$, will of course be $p_1$ and $p_n$, respectively.

[*Proof.* Conditions (i), (ii), (iii), (iv) are clearly satisfied. To verify (v), suppose $1 \le k < n$, and let $P^R = p_n \ldots p_2p_1$ be the mirror-reversal of permutation $P$. Then $r_0[k]$ is null if and only if $k+1$ comes before $k$ in $P$, if and only if $k$ comes after $k+1$ in $P^R$, if and only if $k+1$ comes after $k$ in $P^R$.]

This program shows in fact that *every* twintree arises from some permutation in that way: Given the specification of a twintree, it outputs a permutation $P$ that produces those twins.

Furthermore, the output permutation $P$ will satisfy special conditions. When $k$ is given, let's say that a number $s$ less than $k$ is "small" and a number $l$ greater than $k+1$ is "large". Then

$$\text{if } k \text{ occurs after } k+1\text{, we don't have two consecutive elements } sl \text{ between them;} \qquad (*)$$

$$\text{if } k+1 \text{ occurs after } k\text{, we don't have two consecutive elements } ls \text{ between them.} \qquad (**)$$

In other words, if $k+1$ occurs before $k$ in $P$, any small elements between them must follow any large ones between them $(*)$; otherwise any small elements between them must *precede* any large ones between them $(**)$. A *Baxter permutation* is a permutation that satisfies $(*)$ and $(**)$.

The number of Baxter permutations for $n = 0, 1, 2, 3, 4, 5, \ldots$ turns out to be 1, 1, 2, 6, 22, 92, 422, 2074, 10754, 58202, ... ; in particular, two of the 24 permutations for $n = 4$ do not qualify. (They are the "pi-mutation" 3142 and its reverse, 2413.)

On the other hand we've seen that every permutation does lead to a twintree. Therefore we must be able to get the same twintree from two different permutations. For example, both 3142 and 3412 give the same result when inserted into binary trees, and so do their reverses.

*This program outputs only Baxter permutations.* So it will output 3412 when presented with the twintree defined by 3142.

Historical notes: Twintrees were introduced by Serge Dulucq and Olivier Guibert [*Discrete Math.* **157** (1996), 91–106] in connection with the proof of a conjecture about Young tableaux. Baxter permutations have a more complex history; they're named after Glen Baxter, who considered a related class of permutations while studying the common fixed points of continuous functions from an interval to itself. The story of their naming and their significance in that domain has been well told by William M. Boyce, *Houston Journal of Mathematics* **7** (1981), 175–189.

**2.** The input to this program, in file *stdin*, begins with a line that defines the roots, $t_0$ and $t_1$. The next $n$ lines should then contain five numbers each, namely

$$k \quad l_0[k] \quad r_0[k] \quad l_1[k] \quad r_1[k]$$

for $1 \le k \le n$, in any order. Null links are represented by zero. For example, here's one of the ways to input the twintree defined by 3142:

```
3 2
3 1 4 0 0
1 0 2 0 0
4 0 0 3 0
2 0 0 1 4
```

Incorrect input will be rudely rejected.

The output of this program, in file *stdout*, will be a single line that specifies a Baxter permutation $p_1p_2\ldots p_n$ from which the input could have been obtained. In the example, that output would be '`3 4 1 2`'.

We'll prove below that the answer is unique—or equivalently, that distinct Baxter permutations give distinct twintrees. So we've got a one-to-one correspondence between twintrees and Baxter permutations.

```
#define maxn 1024
#define panic(m, k)
          { fprintf(stderr, "%s!␣(%d)\n", m, k); exit(−666); }
#define pan(m)
          { fprintf(stderr, "%s!\n", m); exit(−66); }
#include <stdio.h>
#include <stdlib.h>
  ⟨Global variables 4⟩;
  ⟨Subroutines 6⟩;
  void main(void)
  {
    register int i, j, k, l, m, n;
    ⟨Input the twintree 3⟩;
    ⟨Check the twintree 5⟩;
    ⟨Output the Baxter permutation 17⟩;
  }
```

**3.** Notice that $r[k]$ is null if and only if $l[k+1]$ is nonnull, in *any* binary tree whose inorder has $k+1$ following $k$. Therefore, using condition (v), $l_0[k]=0$ if and only if $l_1[k]>0$, for $1<k\le n$. Furthermore we always have $l_0[0]=l_1[0]=r_0[n]=r_1[n]=0$. We might as well check those conditions when we read the input.

(If we're interested in saving space, a twintree data structure could be compacted by using just one memory cell to store both $l_0[k]$ and $l_1[k]$, and another to store both $r_0[k]$ and $r_1[k]$, with a single bit to tell us which is null. That's cute, but it would make each access a bit slower.)

```
⟨Input the twintree 3⟩ ≡
  if (fscanf(stdin, "%d␣%d", &t0, &t1) ≠ 2) pan("I␣can't␣read␣the␣root␣numbers");
  for (l = maxn, m = n = 0; fscanf(stdin, "%d", &inx) ≡ 1; n++) {
    if (inx ≤ 0 ∨ inx > l) panic("bad␣index", inx);
    if (inx > m) m = inx;
    if (fscanf(stdin, "%d␣%d␣%d␣%d", &l0[inx], &r0[inx], &l1[inx], &r1[inx]) ≠ 4)
      panic("I␣can't␣read␣l0,r0,l1,r1", inx);
    if (l0[inx] < 0 ∨ l0[inx] > l) panic("l0␣out␣of␣range", inx);
    if (r0[inx] < 0 ∨ r0[inx] > l) panic("r0␣out␣of␣range", inx);
    if (l1[inx] < 0 ∨ l1[inx] > l) panic("l1␣out␣of␣range", inx);
    if (r1[inx] < 0 ∨ r1[inx] > l) panic("r1␣out␣of␣range", inx);
    if (l0[inx] ≥ inx ∨ l1[inx] ≥ inx) panic("l0␣or␣l1␣too␣big", inx);
    if ((r0[inx] ∧ r0[inx] ≤ inx) ∨ (r1[inx] ∧ r1[inx] ≤ inx)) panic("r0␣or␣r1␣too␣small", inx);
    if (l0[inx] ≠ 0 ∧ l1[inx] ≠ 0) panic("l0␣and␣l1␣overlap", inx);
    if (r0[inx] ≠ 0 ∧ r1[inx] ≠ 0) panic("r0␣and␣r1␣overlap", inx);
    if (r0[inx] ≡ r1[inx]) l = inx;    /* this inx should be the final n */
  }
  if (m < n) panic("too␣many␣lines␣of␣input", n − m);
  if (m > n) panic("too␣few␣lines␣of␣input", m − n);
  if (l ≠ n) pan("r0␣and␣r1␣zero␣before␣item␣n");    /* it's not easy to get that error! */
```

This code is used in section 2.

**4.** ⟨Global variables 4⟩ ≡

```
  int inx;    /* data input with fscanf */
  int t0, t1;    /* the roots of T_0 and T_1 */
  int l0[maxn + 1], r0[maxn + 1], l1[maxn + 1], r1[maxn + 1];    /* the links */
```

See also section 18.

This code is used in section 2.

**5.** We must verify that the arrays $l_0$, $r_0$, $l_1$, $r_1$ define binary trees whose nodes are 1, 2, …, $n$ in symmetric order. This is textbook stuff—and fun, because it isn't quite as simple as it may seem at first! We've got to make sure that bad input doesn't get us into an infinite loop, which might happen for example if $l[k]=j$ and $r[j]=k$.

```
⟨Check the twintree 5⟩ ≡
  checkinorder0(t0, 1, n);
  checkinorder1(t1, 1, n);
```

This code is used in section 2.

**6.** ⟨ Subroutines 6 ⟩ ≡

```
void checkinorder0(int root, int lb, int ub)
{
  if (l0[root] ≡ 0) {
    if (root ≠ lb) panic("inorder0␣fails␣left", root);
  } else {
    if (l0[root] < lb) panic("inorder0␣off␣left", root);
    checkinorder0(l0[root], lb, root − 1);
  }
  if (r0[root] ≡ 0) {
    if (root ≠ ub) panic("inorder0␣fails␣right", root);
  } else {
    if (r0[root] > ub) panic("inorder0␣off␣right", root);
    checkinorder0(r0[root], root + 1, ub);
  }
}
void checkinorder1(int root, int lb, int ub)
{
  if (l1[root] ≡ 0) {
    if (root ≠ lb) panic("inorder1␣fails␣left", root);
  } else {
    if (l1[root] < lb) panic("inorder1␣off␣left", root);
    checkinorder1(l1[root], lb, root − 1);
  }
  if (r1[root] ≡ 0) {
    if (root ≠ ub) panic("inorder1␣fails␣right", root);
  } else {
    if (r1[root] > ub) panic("inorder1␣off␣right", root);
    checkinorder1(r1[root], root + 1, ub);
  }
}
```

This code is used in section 2.

**7. Handy facts about Baxter permutations.** As above, let $P$ be the permutation $p_1p_2\ldots p_n$, and let $T_0$ and $T_1$ be the twintrees that result by inserting the elements of $P$ and its reflection $P^R = p_n \ldots p_2p_1$ into an initially empty binary tree.

Then the twintrees obtained from $P^R$ are obviously $T_1$ and $T_0$. We can express that condition algebraically by writing

$$T_\theta(P^R) = T_{\bar\theta}(P).$$

And it's easy to see, directly from the definition, that $P$ is Baxter if and only if $P^R$ is Baxter: Condition $(*)$ for $P$ is condition $(**)$ for $P^R$, and vice versa.

**8.** The *complement* of $P$ is $P^C = \bar p_1\bar p_2 \ldots \bar p_n = (n{+}1{-}p_1)(n{+}1{-}p_2)\ldots(n{+}1{-}p_n)$, obtained by swapping $1 \leftrightarrow n$, $2 \leftrightarrow n-1$, etc.

The twintrees corresponding to $P^C$ are clearly obtained by reversing the roles of left and right—reflecting each tree. That is, when $l[k] = j$ in $T$, we have $r[\bar k] = \bar\jmath$ in the reflected tree $T^R$; similarly, $r[k] = j$ in $T$ implies $l[\bar k] = \bar\jmath$ in $T^R$. Thus we can write

$$T_\theta(P^C) = T_\theta(P)^R.$$

Again, $P$ is Baxter if and only if $P^C$ is Baxter—because complementation, like reflection, interchanges conditions $(*)$ and $(**)$. (Notice that $\bar k = \overline{k+1} + 1$.)

**9.** The *inverse* of $P$, namely $P^- = q_1q_2\ldots q_n$ where $p_j = k \iff q_k = j$, is of course a third basic operation that takes permutations into permutations. This operation is important to us because of the following basic "principle of afterness":

$$p_k > p_l \iff k \text{ comes after } l \text{ in } P^-; \qquad (\dagger)$$
$$k \text{ comes after } l \text{ in } P \iff q_k > q_l. \qquad (\ddagger)$$

**10.** Indeed, the definition of Baxter-hood can be restated nicely in terms of $p$'s and $q$'s: A permutation is Baxter if and only if it doesn't have indices $k$ and $l$ such that

$$q_{k+1} < l \text{ and } q_k > l+1 \text{ and } p_l < k \text{ and } p_{l+1} > k+1; \qquad (*)$$
$$q_k < l \text{ and } q_{k+1} > l+1 \text{ and } p_l > k+1 \text{ and } p_{l+1} < k. \qquad (**)$$

It follows that $P$ is Baxter if and only if $P^-$ is Baxter, because interchanging $q \leftrightarrow p$ and $k \leftrightarrow l$ interchanges $(*) \leftrightarrow (**)$.

**11.** Inversion interacts with reflection and complementation in simple ways, because we have

$$P^{R-} = P^{-C} \qquad \text{and} \qquad P^{RC} = P^{CR}$$

for all permutations $P$. Thus we get up to eight different permutations from any given $P$, namely

$$P,\ P^R,\ P^C,\ P^{RC},\ P^-,\ P^{-R},\ P^{-C},\ P^{-RC},$$

but no more. Sometimes only four of these eight are distinct, as when $P = P^-$ or $P^R = P^C$. In fact, sometimes only two of them are distinct; we have $P^R = P^C = P-$ in cases like $P = 3142$ or $41352$.

But inversion doesn't affect twintrees in any simple way. Indeed, we can't determine the twintrees for $P^-$ from the twintrees for $P$: Both 3142 and 3412 yield the same twins, but 3412 is its own inverse.

**12.** A Baxter permutation remains Baxter if we remove its largest element. (That's easy to check, because any new occurrences of $sl$ or $ls$ in $(*)$ or $(**)$ would have been $snl$ or $lns$ before $n$ is removed.) Therefore we can obtain all of the Baxter permutations on $\{1, 2, \ldots, n\}$ from those on $\{1, \ldots, n-1\}$, if we insert $n$'s appropriately.

Consider, for example, the permutation 21836745, which satisfies both $(*)$ and $(**)$. We can insert 9 into it in nine ways:

$$921836745, 291836745, 219836745, 218936745, 218396745, 218369745, 218367945, 218367495, 218367459.$$

Which of these extensions retains its Baxterhood?

Well, the ones that fail are 291836745 ($k = 2$), 218396745 ($k = 7$), 218369745 ($k = 7$), and 218367495 ($k = 5$). A bit of thought reveals the general rule: *We can Baxterly place $n$ within a Baxter permutation of order $n-1$ if and only if we put it just before a left-to-right maximum, or just after a right-to-left maximum.* For example, the left-to-right maxima in our example are 2 and 8; the right-to-left maxima are 5, 7, 8; we successfully placed 9 before the 2, before the 8, after the 5, after the 7, and after the 8.

Here's the proof. For convenience let's call a left-to-right maximum an "LRmax," etc. The inserted element $n$ will mess up condition $(*)$ if and only if we place it just after a small element $s$ between $k+1$ and $k$. That $s$ wasn't an RLmax, because $s < k$; nor was $s$ immediately followed by an LRmax, because $(*)$ was true. The inserted $n$ will mess up $(**)$ if and only if we place it immediately before a small element $s$ between $k$ and $k+1$. That $s$ wasn't an LRmax, because $s < k$; nor was $s$ immediately preceded by an RLmax, because $(**)$ was true. In either case, failure occurs if and only if we haven't allowed that position.

**13.** A Baxter permutation remains Baxter if we remove its final element $p_n$ and subtract 1 from each $p_k$ for which $k < n$ and $p_k > p_n$. Indeed, this operation is equivalent to deleting $n$ from the inverse permutation!

For example, we've seen that 218367945 is a Baxter permutation. Its inverse, 214895637, is therefore also Baxter. Removing 9 gives the Baxter permutation 21485637, whose inverse 21735684 is Baxter.

**14.** Similarly, we can remove 1 and decrease each remaining element by 1.

We can also remove $p_1$, and renumber the others.

Indeed, a moment's thought shows also the converse: If $P$ is non-Baxter, we can use some combination of the operations remove-the-largest, remove-the-last, remove-the-smallest, and/or remove-the-first, until we reach either 3142 or 2413.

**15. Solving the problem.** Instead of renumbering, after the deletion of $n$ or $p_n$ or 1 or $p_1$ from a Baxter permutation, we can simply consider the remaining sequence to be a permutation of the numbers that are left; and we can form a twintree with them, ignoring the tree links from nodes that have been removed.

For example, we can regard '2763' as a permutation of the elements $\{2,3,6,7\}$. Tree $T_0$ of its twintree structure has root 2 and links

$$l_0[2]=0,\ r_0[2]=7;\ \ l_0[3]=0,\ r_0[3]=0;\ \ l_0[6]=3,\ r_0[6]=0;\ \ l_0[7]=6,\ r_0[7]=0;$$

tree $T_1$, similarly, has root 3 and links

$$l_1[2]=0,\ r_1[2]=0;\ \ l_1[3]=2,\ r_1[3]=6;\ \ l_1[6]=0,\ r_1[6]=7;\ \ l_1[7]=0,\ r_1[7]=0.$$

All other links are irrelevant. The inorder of both trees is the natural order of the elements that remain, namely 2367.

Call this a "generalized twintree," for a "generalized permutation." If $k$ is an element of a generalized permutation, the notation '$k+1$' stands not for the sum of $k$ and 1 but rather for the element that immediately follows $k$, namely $k$'s inorder successor. In our example, $3+1=6$ and $6-1=3$.

**16.** Recall that our task is to discover a Baxter permutation that yields a given twintree. We might as well extend that task, by assuming that's we've been given a *generalized* twintree, for which we want to discover a *generalized* Baxter permutation.

Suppose we've been given a generalized twintree with $n$ nodes. The solution is obvious when $n=1$, so we may assume that $n>1$.

The first step is also obvious: We know $p_1$, because it's the root of $T_0$. So our strategy will be to delete $p_1$ from the generalized twintree, then figure out what generalized twintree should produce the other elements $p_2, \ldots, p_n$ of the generalized permutation.

Let $p=p_1$. Notice that $p$ is always a leaf of $T_1$, because it was the last element inserted into that tree. So it's easy to remove $p$ from $T_1$; we simply zero out the link that pointed to it.

There are two cases. If $p$ was a left child, say $l_1[i]=p$, we'll want to set $l_1[i]\leftarrow 0$. In that case $i=p+1$, the inorder successor of $p$. Otherwise $p$ was a right child, say $r_1[i]=p$, and we'll want to set $r_1[i]\leftarrow 0$; in that case $i=p-1$, the inorder *predecessor* of $p$.

How should we remove $p$ from $T_0$? If $l_0[p]=0$, we simply move the root of $T_0$ down to $r_0[p]$. Similarly, if $r_0[p]=0$, we simply move $T_0$'s root to $l_0[p]$.

But if both $j=l_0[p]$ and $k=r_0[p]$ are nonzero, we need to decide which of them should be the new root of $T_0$, depending on which of them came earlier in the permutation we're trying to discover. And we also need to figure out how to merge the other nodes into a single tree.

Fortunately there's only one way to go. Suppose, for instance, that $i=p+1$; the other case is symmetrical. Since $j<p$, the element $p-1$ must exist. And since $p$ is $l_1[i]$, $p-1$ must have occurred after $i$ in the permutation.

We now can prove that $j$ itself, and all of its descendants, follow $i$. Otherwise we'd have $p-1$ preceded by $p+1$, preceded by some element $s$ less than $p$, preceded by $p$, contradicting $(*)$ (with $k=p-1$ and $l=p+1$).

In particular, the new root of $T_0$ must be $k$.

Finally, since $i=p+1$, we must have $l_0[i]=0$. Therefore we set $l_0[i]\leftarrow j$; this properly reflects the fact that $j$ and all of its descendants follow $i$.

(This stunning construction is essentially Lemma 4 of Dulucq and Guibert's paper, although they worked with inverse permutations and "increasing binary trees" instead of with binary search trees.)

**17.** So here's the algorithm that the argument above has forced upon us.

We start by making a table of everybody's parent in $T_1$, setting $parent[p] = i$ if $l_1[i] = p$, but $parent[p] = -i$ if $r_1[i] = p$.

```
⟨ Output the Baxter permutation 17 ⟩ ≡
  for ($k = 1$; $k \le n$; $k$++) {
    if ($l1[k]$) $parent[l1[k]] = k$;
    if ($r1[k]$) $parent[r1[k]] = -k$;
  }
  while (1) {
    printf("%d␣", t0);    /* the first element, $p$, of the remaining generalized perm */
    $i = parent[t0]$;
    if ($i > 0$) {    /* $i = p + 1$, the case considered above */
      $l1[i] = 0$;    /* remove $p$ from $T_1$ */
      if ($r0[t0] \equiv 0$) $t0 = l0[t0]$;    /* $p$ is the largest that remains */
      else {
        $l0[i] = l0[t0]$;
        $t0 = r0[t0]$;
      }
    } else if ($i \equiv 0$) break;    /* tree size was 1 */
    else {
      $i = -i$;    /* $i = p - 1$, the symmetrical case */
      $r1[i] = 0$;    /* remove $p$ from $T_1$ */
      if ($l0[t0] \equiv 0$) $t0 = r0[t0]$;    /* $p$ is the smallest that remains */
      else {
        $r0[i] = r0[t0]$;
        $t0 = l0[t0]$;
      }
    }
  }
  printf("\n");
```

This code is used in section 2.

**18.** ⟨ Global variables 4 ⟩ +≡

```
  int $parent[maxn + 1]$;    /* parents in $T_1$ */
```

## 19. Index.

# TWINTREE-TO-BAXTER