**1. Intro.** This (hastily written) program computes a floorplan that corresponds to a given Baxter permutation. 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.

The input permutation is supposed to 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 $(**)$.

Let's call the given Baxter permutation $P = p_1 p_2 \dots p_n$. We'll construct a floorplan whose rooms are the numbers $\{1, 2, \dots, n\}$. The diagonal order of those rooms will be simply $12 \dots n$; and their antidiagonal order will be $p_1 p_2 \dots p_n$.

Floorplans have an interesting "four-way" order, under which any two distinct rooms $j$ and $k$ are in exactly one of four relationships to each other: Either $j$ is left of $k$ (written $j \Rightarrow k$), or $j$ is above $k$ (written $j \Downarrow k$), or $j$ is right of $k$ (written $j \Leftarrow k$), or $j$ is below $k$ (written $j \Uparrow k$ <). The diagonal order is the linear order "above or left"; the antidiagonal order is the linear order "below or left".

Therefore we must have the following (nice) situation:

$$\begin{aligned} j \Rightarrow k &\iff j < k \text{ and } j \text{ precedes } k \text{ in } P; \\ j \Downarrow k &\iff j < k \text{ and } j \text{ follows } k \text{ in } P; \\ j \Leftarrow k &\iff j > k \text{ and } j \text{ follows } k \text{ in } P; \\ j \Uparrow k &\iff j > k \text{ and } j \text{ precedes } k \text{ in } P. \end{aligned}$$

Furthermore, $j$ precedes $k$ in $P$ if and only $q_j < q_k$, where $q_1 q_2 \dots q_n$ is $P^-$, the inverse of permutation $P$.

Any permutation $P$ defines a four-way order, according to those rules. But only a Baxter permutation defines the four-way order derivable from a floorplan. For example, the "pi-mutation" 3142 defines the four-way order with 1 left of 2, 1 above 3, 1 left of 4, 2 above 3, 2 above 4, 3 left of 4; that can happen in a floorplan only if there's at least one more room. (For instance, we could put "room 2.5" to the right of 1, below 2, above 3, and left of 4. The Baxter permutation for that floorplan would be 3 1 2.5 4 2.)

The rooms of a floorplan are delimited by horizontal and vertical line segments called "bounds," which don't intersect each other. The number of horizontal bounds in the floorplan we shall output is two more than the number of *descents* in $P$ (that is, places where $p_k > p_{k+1}$); and the number of vertical bounds is two more than the number of *ascents* (where $p_k < p_{k+1}$).

By the way, changing $P$ to $P^R$ corresponds to transposing the floorplan about its main diagonal. Changing $P$ to $P^C$ corresponds to transposing the floorplan about its other diagonal. Changing $P$ to $P^-$ corresponds to a top-bottom reflection of the floorplan. Thus the eight Baxter permutations obtained from $P$ by reflection, complementation, and/or inversion correspond to the eight standard "isometric" transformations that can be made to floorplans.

**2.** The input permutation appears in *stdin*, as the sequence of numbers $p_1\ p_2\ \ldots\ p_n$ (separated by whitespace). The output floorplan will be a specification that conforms to the input conventions of the companion program FLOORPLAN-TO-TWINTREE, with the rooms in ascending order.

```
#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 ⟩;
  void main(void)
  {
    register int i, j, k, l, m, n;
    ⟨ Input the permutation 3 ⟩;
    ⟨ Check for Baxterhood 5 ⟩;
    ⟨ Compute the floorplan 6 ⟩;
    ⟨ Output the floorplan 10 ⟩;
  }
```

**3.** ⟨ Input the permutation 3 ⟩ ≡

```
  for (m = n = 0; fscanf(stdin,"%d",&inx) ≡ 1; n++) {
    if (inx ≤ 0 ∨ inx > maxn) panic("element␣out␣of␣range",inx);
    if (inx > m) m = inx;
    p[n + 1] = inx;
  }
  if (m > n) panic("too␣few␣elements",m − n);
  if (m < n) panic("too␣many␣elements",n − m);
  for (k = 1; k ≤ n; k++) q[p[k]] = k;      /* compute the inverse */
  for (k = 1; k ≤ n; k++)
    if (q[k] ≡ 0) panic("missing␣element",k);
```

This code is used in section 2.

**4.** ⟨ Global variables 4 ⟩ ≡

```
  int inx;      /* data input with fscanf */
  int p[maxn + 1], q[maxn + 1];
```

See also section 9.

This code is used in section 2.

**5.** The following check might take quadratic time, because I tried to make it as simple as possible.

If you want to test the Baxter property in linear time, there's a tricky way to do it: (1) Feed the permutation $P$ to OFFLINE-TREE-INSERTION. (2) Also feed its reflection, $P^R$, to OFFLINE-TREE-INSERTION. (3) Edit those two outputs to make a twintree and feed that twintree to TWINTREE-TO-BAXTER. (4) Compare that result to $P$. If $P$ is Baxter, you'll get it back again. [An almost equivalent method was in fact published by Johnson M. Hart, *International Journal of Computer and Information Sciences* **9** (1980), 307–321, and it's instructive to compare the two approaches.]

[If you omit this check, you'll get a floorplan whose anti-diagonal permutation is $P$. But if $P$ isn't Baxter, the *diagonal* permutation of that floorplan won't be 1 2 ... $n$.]

```
⟨ Check for Baxterhood 5 ⟩ ≡
  for (k = 2; k < n − 1; k++) {
    if (q[k] < q[k + 1]) {
      for (l = q[k] + 1; l < q[k + 1] − 1; l++)
        if (p[l] > k ∧ p[l + 1] < k) panic("not␣Baxter␣**", k);
    } else {
      for (l = q[k + 1] + 1; l < q[k] − 1; l++)
        if (p[l] < k ∧ p[l + 1] > k) panic("not␣Baxter␣*", k);
    }
  }
```

This code is used in section 2.

**6. The key algorithm.** We get to use a particularly nice method here, thanks to the insights of Eyal Ackerman, Gill Barequet, and Ron Y. Pinter [*Discrete Applied Mathematics* **154** (2006), 1674–1684]. The four bounds *lft*, *bot*, *rt*, and *top* of each room can be filled in systemically as we march through $P$, taking linear time because we spend only a small bounded number of steps between the times when we make a contribution to the final plan.

The algorithm maintains two stacks, *RLmin* and *RLmax*, which record the current right-to-left minima and maxima in the permutation read so far. The rooms on *RLmin* are precisely those for which *lft* and *bot* have been filled, but not yet *top*. The rooms on *RLmax* are precisely those for which *lft* and *bot* have been filled, but not yet *rt*.

Values in the *bot* and *top* arrays are indices of horizontal bounds; values in the *lft* and *rt* arrays are indices of vertical bounds.

At the end, we needn't fill in the missing values of *rt* and *top*, because they are zero (and that's what we want).

This algorithm is almost too good to be true! It's valid, however, because it can be seen to create the antidiagonal floorplan, step by step.

```
⟨ Compute the floorplan 6 ⟩ ≡
  minptr = maxptr = 1, RLmin[0] = RLmax[0] = j = p[1], lft[j] = bot[j] = n;
  for (k = 1; k < n; k++) {
    i = p[k], j = p[k + 1];      /* i is at the top of both RLmin and RLmax */
    if (i < j) ⟨ Create a new vertical bound 7 ⟩
    else ⟨ Create a new horizontal bound 8 ⟩;
  }
```

This code is used in section 2.

**7.**

```
⟨ Create a new vertical bound 7 ⟩ ≡
  {
    lft[j] = rt[i] = n − k, maxptr−−, RLmin[minptr++] = j;
    while (maxptr ∧ RLmax[maxptr − 1] < j) rt[RLmax[−−maxptr]] = n − k;
    bot[j] = (maxptr ? top[RLmax[maxptr − 1]] : n);
    RLmax[maxptr++] = j;
  }
```

This code is used in section 6.

**8.**

```
⟨ Create a new horizontal bound 8 ⟩ ≡
  {
    bot[j] = top[i] = n − k, minptr−−, RLmax[maxptr++] = j;
    while (minptr ∧ RLmin[minptr − 1] > j) top[RLmin[−−minptr]] = n − k;
    lft[j] = (minptr ? rt[RLmin[minptr − 1]] : n);
    RLmin[minptr++] = j;
  }
```

This code is used in section 6.

**9.**

```
⟨ Global variables 4 ⟩ +≡
  int lft[maxn + 1], bot[maxn + 1], rt[maxn + 1], top[maxn + 1];
  int RLmin[maxn], RLmax[maxn];      /* the stacks */
  int minptr, maxptr;      /* the current stack sizes */
```

**10.**

```
⟨ Output the floorplan 10 ⟩ ≡
  for (k = 1; k ≤ n; k++) printf("%d␣y%d␣y%d␣x%d␣x%d\n", k, n − top[k], n − bot[k], n − lft[k], n − rt[k]);
```

This code is used in section 2.

## 11. Index.

# BAXTER-TO-FLOORPLAN