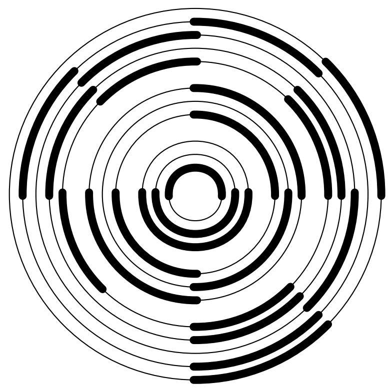
(See https://cs.stanford.edu/~knuth/programs.html for date.)

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, ..., 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 $d_0, d_1, \ldots, d_{2n}$ 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 $d_{p+1}, \ldots, d_{2n}$ 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 $r \ge 2$.) 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 check(s)
    { y -= d[--p];
      if (d[p] ≠ s) fprintf(stderr, "Rejection at position %d, case %d!\n", p, serial); }
```

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

This code is used in section 2.

11.

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

This code is used in section 10.

12.

```
⟨ Remove a wall's base 12 ⟩ ≡
  {
    for (k = 0; k ≤ ww; k += 2) ring[m][k + 1] = 0;
    m−−, ww >>= 1, w−−;
    y = ww, mode = −1;
  }
```

This code is used in section 11.

13. If $y = 0$ and $mode > 0$, 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 $mode < 0$, there must be a free brick in segment *ww* at this time. (Think about it.)

```
⟨ Change the mode and goto look 13 ⟩ ≡
  {
    mode = −mode; goto look;
  }
```

This code is used in section 11.

14. Index.

check: 10, 11.
count: 2, 5.
d: 2.
done: 2, 4.
fprintf: 5, 10.
i: 2.
j: 2.
k: 2.
look: 11, 13.
m: 2.
main: 2.
mode: 2, 7, 9, 10, 11, 12, 13.
n: 2.
nn: 2, 3, 4, 5, 10.
notfound: 11.
p: 2.
printf: 2.
ring: 2, 7, 8, 9, 10, 11, 12.
ringcount: 2, 7, 9, 11.
serial: 2, 5, 10.
stderr: 5, 10.
w: 2.
wall: 2, 7, 8, 10, 11.
ww: 2, 7, 8, 9, 10, 11, 12, 13.
x: 2.
y: 2.

⟨ Begin a new wall 8 ⟩ Used in section 7.
⟨ Change the mode and **goto** *look* 13 ⟩ Used in section 11.
⟨ Check the inverse bijection 10 ⟩ Used in section 2.
⟨ Check the number of walls 5 ⟩ Used in section 2.
⟨ Find the tower corresponding to d 7 ⟩ Used in section 2.
⟨ Move to the next nested string, or **goto** *done* 4 ⟩ Used in section 2.
⟨ Place a brick 9 ⟩ Used in section 7.
⟨ Remove a brick if it's free and ready, or $check(-mode)$ 11 ⟩ Used in section 10.
⟨ Remove a wall's base 12 ⟩ Used in section 11.
⟨ Set up the first nested string, d 3 ⟩ Used in section 2.

VIENNOT

	Section	Page
Introduction	1	1
The main algorithm	6	4
The inverse algorithm	10	6
Index	14	7