

November 24, 2020 at 13:25

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

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 k th 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 k th wall are identified by a sequence of segment numbers p_1, \dots, p_t , where $1 \leq p_1 < \dots < p_t \leq 2^k$. 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, \dots by proceeding counterclockwise from there.) These sequences must satisfy three constraints:

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

2. And what is a nested string? A nested string (aka Dyck word) of order n is a sequence $d_0, d_1, \dots, d_{2n-1}$ of ± 1 s whose partial sums $y_k = d_0 + \dots + d_k$ are nonnegative, and whose overall sum y_{2n} is zero. Its height is $\max_{0 \leq k < 2n} y_k$. 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  $\pm 1$ s */
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_Keppler_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 ⟩ \equiv
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 ⟩ \equiv
 $d[i] = -1$;
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 ($j \equiv 0$) **goto** *done*;
 }
 $d[j] = +1, i = nn - 2$;
}

This code is used in section 2.

5. ⟨ Check the number of walls 5 ⟩ \equiv
for ($m = j = k = 1$; $k < nn - 1$; $j += d[k], k++$)
 if ($j \geq ((1 \ll m) - 1)$) $m++$;
 $m--$; /* now there should be m walls */
 $count[m]++, serial++$;
if ($w \neq m$) {
 $fprintf(stderr, "I_\square goofed_\square on_\square case_\square \%d.\backslash n", serial)$;
}

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 k th wall of the corresponding Kepler tower at the moment the partial sum $d_0 + \dots + d_p$ 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 ± 1 s 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, \dots, 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 $d_1 + \dots + d_{2n} = -2$.

The k th 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}, \dots, 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 $d_0 + \dots + d_p = 7$.

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 $(d_0, d_1, \dots, d_{2n})$ 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 $ring[0], ring[1], \dots, ring[m]$, 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 k th wall consists of $ring[j]$ for $wall[k - 1] \leq j < wall[k]$, for $1 \leq k \leq w$. If we have most recently looked at $d[p]$, variable y is the partial sum $d[p' + 1] + \dots + d[p]$ 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 ( $y = p = 0; y \neq -ww;$ ) {
    if ( $y \equiv 0$ )  $mode = -d[++p], y -= mode;$ 
    else if ( $y \equiv ww$ ) ⟨Begin a new wall 8⟩
    else {
       $y += d[++p];$ 
      if ( $d[p] \equiv mode$ ) ⟨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 ( $k = 0; k \leq ww; k += 2$ )  $ring[m][k + 1] = 1;$ 
     $y = 0;$ 
  }

```

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 ( $j = m; ring[j][k - 1] \equiv 0 \wedge ring[j][k] \equiv 0 \wedge ring[j][k + 1] \equiv 0; j--$ ) ;
    if ( $j \equiv m$ )  $m++;$  /* enter a new ring, initially empty */
     $ring[j + 1][k] = 1;$ 
    if ( $k \equiv 1$ )  $ring[j + 1][ww + 1] = 1;$ 
    else if ( $k \equiv ww$ )  $ring[j + 1][0] = 1;$ 
     $ringcount[j + 1]++;$ 
  }

```

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

	1	2
	3	4
	5	6
	7	8
	9	10
	11	12
	13	14
	15	16
	17	18
	19	20
	21	22
	23	24
	25	26
	27	28
	29	30
	31	32
	33	34
	35	36
	37	38
	39	40
	41	42
	43	44
	45	46
	47	48
	49	50
	51	52
	53	54
	55	56
	57	58
	59	60
	61	62
	63	64
	65	66
	67	68
	69	70
	71	72
	73	74
	75	76
	77	78
	79	80
	81	82
	83	84
	85	86
	87	88
	89	90
	91	92
	93	94
	95	96
	97	98
	99	100
	101	102
	103	104
	105	106
	107	108
	109	110
	111	112
	113	114
	115	116
	117	118
	119	120
	121	122
	123	124
	125	126
	127	128
	129	130
	131	132
	133	134
	135	136
	137	138
	139	140
	141	142
	143	144
	145	146
	147	148
	149	150
	151	152
	153	154
	155	156
	157	158
	159	160
	161	162
	163	164
	165	166
	167	168
	169	170
	171	172
	173	174
	175	176
	177	178
	179	180
	181	182
	183	184
	185	186
	187	188
	189	190
	191	192
	193	194
	195	196
	197	198
	199	200
	201	202
	203	204
	205	206
	207	208
	209	210
	211	212
	213	214
	215	216
	217	218
	219	220
	221	222
	223	224
	225	226
	227	228
	229	230
	231	232
	233	234
	235	236
	237	238
	239	240
	241	242
	243	244
	245	246
	247	248
	249	250
	251	252
	253	254
	255	256
	257	258
	259	260
	261	262
	263	264
	265	266
	267	268
	269	270
	271	272
	273	274
	275	276
	277	278
	279	280
	281	282
	283	284
	285	286
	287	288
	289	290
	291	292
	293	294
	295	296
	297	298
	299	300
	301	302
	303	304
	305	306
	307	308
	309	310
	311	312
	313	314
	315	316
	317	318
	319	320
	321	322
	323	324
	325	326
	327	328
	329	330
	331	332
	333	334
	335	336
	337	338
	339	340
	341	342
	343	344
	345	346
	347	348
	349	350
	351	352
	353	354
	355	356
	357	358
	359	360
	361	362
	363	364
	365	366
	367	368
	369	370
	371	372
	373	374
	375	376
	377	378
	379	380
	381	382
	383	384
	385	386
	387	388
	389	390
	391	392
	393	394
	395	396
	397	398
	399	400
	401	402
	403	404
	405	406
	407	408
	409	410
	411	412
	413	414
	415	416
	417	418
	419	420
	421	422
	423	424
	425	426
	427	428
	429	430
	431	432
	433	434
	435	436
	437	438
	439	440
	441	442
	443	444
	445	446
	447	448
	449	450
	451	452
	453	454
	455	456
	457	458
	459	460
	461	462
	463	464
	465	466
	467	468
	469	470
	471	472
	473	474
	475	476
	477	478
	479	480
	481	482
	483	484
	485	486
	487	488
	489	490
	491	492
	493	494
	495	496
	497	498
	499	500
	501	502
	503	504
	505	506
	507	508
	509	510
	511	512
	513	514
	515	516
	517	518
	519	520
	521	522
	523	524
	525	526
	527	528
	529	530
	531	532
	533	534
	535	536
	537	538
	539	540
	541	542
	543	544
	545	546
	547	548
	549	550
	551	552
	553	554
	555	556
	557	558
	559	560
	561	562
	563	564
	565	566
	567	568
	569	570
	571	572
	573	574
	575	576
	577	578
	579	580
	581	582
	583	584
	585	586
	587	588
	589	590
	591	592
	593	594
	595	596
	597	598
	599	600
	601	602
	603	604
	605	606
	607	608
	609	610
	611	612
	613	614
	615	616
	617	618
	619	620
	621	622
	623	624
	625	626
	627	628
	629	630
	631	632
	633	634
	635	636
	637	638
	639	640
	641	642
	643	644
	645	646
	647	648
	649	650
	651	652
	653	654
	655	656
	657	658
	659	660
	661	662
	663	664
	665	666
	667	668
	669	670
	671	672
	673	674
	675	676
	677	678
	679	680
	681	682
	683	684
	685	686
	687	688
	689	690
	691	692
	693	694
	695	696
	697	698
	699	700
	701	702
	703	704
	705	706
	707	708
	709	710
	711	712
	713	714
	715	716
	717	718
	719	720
	721	722
	723	724
	725	726
	727	728
	729	730
	731	732
	733	734
	735	736
	737	738
	739	740
	741	742
	743	744
	745	746
	747	748
	749	750
	751	752
	753	754
	755	756
	757	758
	759	760
	761	762
	763	764
	765	766
	767	768
	769	770
	771	772
	773	774
	775	776
	777	778
	779	780
	781	782
	783	784
	785	786
	787	788
	789	790
	791	792
	793	794
	795	796
	797	798
	799	800
	801	802
	803	804
	805	806
	807	808
	809	810
	811	812
	813	814
	815	816
	817	818
	819	820
	821	822
	823	824
	825	826
	827	828
	829	830
	831	832
	833	834
	835	836
	837	838
	839	840
	841	842
	843	844
	845	846
	847	848
	849	850
	851	852
	853	854
	855	856
	857	858
	859	860
	861	862
	863	864
	865	866
	867	868
	869	870
	871	872
	873	874
	875	876
	877	878
	879	880
	881	882
	883	884
	885	886
	887	888
	889	890
	891	892
	893	894
	895	896
	897	898
	899	900
	901	902
	903	904
	905	906
	907	908
	909	910
	911	912
	913	914
	915	916
	917	918
	919	920
	921	922
	923	924
	925	926
	927	928
	929	930
	931	932
	933	934
	935	936
	937	938
	939	940
	941	942
	943	944
	945	946
	947	948
	949	950
	951	952
	953	954
	955	956
	957	958
	959	960
	961	962
	963	964
	965	966
	967	968
	969	970
	971	972
	973	974
	975	976
	977	978
	979	980
	981	982
	983	984
	985	986
	987	988
	989	990
	991	992
	993	994
	995	996
	997	998
	999	1000
	1001	1002
	1003	1004
	1005	1006
	1007	1008
	1009	1010
	1011	1012
	1013	1014
	1015	1016
	1017	1018
	1019	1020
	1021	1022
	1023	1024
	1025	1026
	1027	1028
	1029	1030
	1031	1032