The tower and marbles puzzle

Rachid Hamadi, CSE, UNSW

COMP9021 Principles of Programming, Term 3, 2019

```
[1]: from math import sqrt, ceil
     from random import randint
```

In order to test the quality of the products of a marble manufacturer, one plans to drop marbles from various levels of a tower and see whether they break. We make the following assumptions:

- If a marble breaks when dropped from a given level, then it would have broken if dropped from a higher level.
- All marbles are of the same quality: they all break or all do not break if dropped from a given level.

One intends to find out the highest level L such that a marble dropped from level L does not break, but breaks if dropped from a higher level, if any. The aim is to minimise the effort, that is:

- find out the maximum number of drops d needed in the worst case to get the correct answer;
- come up with a strategy to, for any marble quality, get the correct answer with no more than d drops.

The problem has two parameters:

- the number of available marbles, m;
- the number of levels, n.

Both m and n are assumed to be nonnegative integers.

The solution is straightforward if m is equal to 1. Then d is equal to n and the only valid strategy consists in dropping the unique marble from level 1, then from level 2, then from level 3... until it breaks, if it ever does. The maximum of n drops is needed if the marbles are of the best quality and do not break when dropped from any level, or if they are of the next best quality and break when dropped from the highest level, but not from any level below.

There exists a general solution for any value of m, but a simpler solution exists in case m is equal to 2; before describing and implementing the former, we describe and implement the latter. So for now, we set the value of m to 2.

Rather than directly computing d from n, we reverse the problem: given d, what is the maximum number of levels n (the maximum height of the tower) such that for any marble quality, one is sure to correctly assess the marbles' quality with no more than d drops? If the first marble ever breaks, dropped from level L, one is left with the second marble only and has no other option but explore all levels below L, if any, starting with the lowest such level, and making one's way up on all unexplored levels below L.

- This means that the first drop should be from level d: if the marble breaks, then there are $d - 1$ drops left and the second marble has to be dropped from level 1, then from level 2... potentially up to level $d - 1$ (in case the lowest level from which marbles break is either level d or level $d - 1$). It is best to make the first drop from as high as possible, so the first drop should be made from level d.

- The same reasoning shows that in case the first marble does not break following the first drop, it should then be dropped from level $d + (d - 1)$: if it breaks, then there are $d - 2$ drops left and there are $d - 2$ unexplored levels between level d, the highest level L for which it is known that a marble dropped from L does not break, and level $d + (d - 1)$, the lowest level L' for which it is known that a marble dropped from L' does break. The second marble should be dropped from level $d + 1$, then from level $d + 2$... potentially up to level $2d - 2$ (in case the lowest level from which marbles break is either level $2d - 1$ or level $2d - 2$).

- If following the second drop, the first marble does not break, the third drop should be from level $d + (d - 1) + (d - 2)$.

- ...

- In case marbles are of the best quality and never break, the first marble only will be used, dropped from level d, then from level $d + (d - 1)$, then from level $d + (d - 1) + (d - 2)$... and eventually, for the dth drop, from level $d + (d - 1) + (d - 2) + \cdots + 2 + 1$.

This shows that n is equal to $\frac{d(d+1)}{2}$; n and d satisfy the equality $d^2 + d - 2n = 0$.

Of course, only some nonnegative integers n are of the form $\frac{d(d+1)}{2}$ for some nonnegative integers d. In the original problem, n is known and arbitrary, while d has to be computed from n. We infer from the previous considerations that d is the smallest nonnegative integer such that $d^2 + d - 2n \geq 0$, hence $d = \lceil \frac{-1+\sqrt{1+8n}}{2} \rceil$.

- Assume that $n = 6$; then $d = 3$. The first drop should be from level 3, and if the marble does not break, the second drop should be from level $3 + 2 = 5$, and if the marble still does not break, the third drop should be from level $3 + 2 + 1 = 6$.

- Assume that $n = 5$; then $d = 3$ too. The first drop should be from level 3, and if the marble does not break, the second drop should be from level $3 + 2 = 5$, and if the marble does not break, we are done, as there is no higher level to explore.

- Assume that $n = 4$; then $d = 3$ still. The first drop should be from level 3. If the marble does not break, the second drop could afford to be from level $3 + 2 = 5$, but since there is no such level, it should be from level 4.

This illustrates that given i between 1 and d, there can only be an ith drop with the first marble in case n is at least equal to $1 + \Sigma_{j=0}^{i-2}(d - j)$, and if there is an ith drop with the first marble, then it should be from level $\min(n, \Sigma_{j=0}^{i-1}(d - j))$.

We define a variable, `breaking_level`, meant to denote the value of the lowest level L such that a marble dropped from L breaks, if there is such a level. We should also allow for the case where the marble is of best quality and does not break when dropped from any level. To simplify the design of the solution, it is best to let $n + 1$ be the corresponding value. One can think of $n + 1$ to be an additional, "virtual" level, such that any marble dropped from that level is sure to break, which is consistent with the reasonable assumption that a marble is sure to break when dropped from high enough.

We define two variables, `low` and `high`, meant to keep track of the highest level L for which it is known that a marble does not break when dropped from level L, and of the lowest level L'

for which it is known that a marble does break when dropped from level L', respectively. The initialisation should capture what we know before the procedure starts: `low` should be set to `0` and `high` to $n + 1$. At any stage of the computation, `low` will be smaller than `high`.

There is some uncertainty for as long as there is a gap between `low` and `high`, so for as long as `low` is strictly smaller than `high` - 1. The next drop will then be from level l for some l strictly between `low` and `high`, raising the value of `low` to l in case the marble does not break, decreasing the value of `high` to l in case the marble breaks, in both cases, reducing the gap between `low` and `high`.

- When using the first marble, the first drop should be d levels above the current value of `low`, set to `0`, the second drop should be $d - 1$ levels above the current value of `low`, changed to d after the first drop, the third drop should be $d - 2$ levels above the current value of `low`, changed to $d + (d - 1)$ after the second drop... in each case unless the computed value is at least equal to the value of `high`, as it should then be set to `high` - 1.
- When using the second marble, every drop should be 1 level above the current value of `low`.

A variable `step` can keep track of by how many levels above the current value of `low` to go up for the next drop.

After the last drop, `high` is equal to `low` + 1, and it is known that:

- if `high` is equal to $n + 1$, then marbles are of best quality and do not break;
- if `high` is strictly smaller than $n + 1$, then a marble breaks when dropped from level `high`, but not below.

The following function puts together all observations that precede, and allows one to systematically test the strategy, in all cases, for a given value of n:

```
[2]: def systematic_two_marbles_simulation(n):
         d = ceil((sqrt(8 * n + 1) - 1) / 2)
         if d == 0:
             print('No drop will be needed.')
         elif d == 1:
             print('At most 1 drop will be needed.')
         else:
             print('At most', d, 'drops will be needed.')
         for breaking_level in range(1, n + 2):
             if breaking_level <= n:
                 print('\nStrategy for marbles breaking on level '
                       f'{breaking_level} and not below:'
                      )
             else:
                 print('\nStrategy for marbles not breaking:')
             low = 0
             high = n + 1
             step = d
             while low < high - 1:
                 level = min(low + step, high - 1)
                 if breaking_level <= level:
                     print(f'From level {level}... marble breaks!')
```

```
                high = level
                step = 1
        else:
            print(f'From level {level}... marble does not break!')
            low = level
            # If we are using the second marble, step is equal to 1
            # and should remain equal to 1.
            if step > 1:
                step -= 1
```

[3]: `systematic_two_marbles_simulation(1)`

At most 1 drop will be needed.

Strategy for marbles breaking on level 1 and not below:
From level 1... marble breaks!

Strategy for marbles not breaking:
From level 1... marble does not break!

[4]: `systematic_two_marbles_simulation(2)`

At most 2 drops will be needed.

Strategy for marbles breaking on level 1 and not below:
From level 2... marble breaks!
From level 1... marble breaks!

Strategy for marbles breaking on level 2 and not below:
From level 2... marble breaks!
From level 1... marble does not break!

Strategy for marbles not breaking:
From level 2... marble does not break!

[5]: `systematic_two_marbles_simulation(3)`

At most 2 drops will be needed.

Strategy for marbles breaking on level 1 and not below:
From level 2... marble breaks!
From level 1... marble breaks!

Strategy for marbles breaking on level 2 and not below:
From level 2... marble breaks!
From level 1... marble does not break!

```
Strategy for marbles breaking on level 3 and not below:
From level 2... marble does not break!
From level 3... marble breaks!

Strategy for marbles not breaking:
From level 2... marble does not break!
From level 3... marble does not break!
```

[6]: `systematic_two_marbles_simulation(4)`

```
At most 3 drops will be needed.

Strategy for marbles breaking on level 1 and not below:
From level 3... marble breaks!
From level 1... marble breaks!

Strategy for marbles breaking on level 2 and not below:
From level 3... marble breaks!
From level 1... marble does not break!
From level 2... marble breaks!

Strategy for marbles breaking on level 3 and not below:
From level 3... marble breaks!
From level 1... marble does not break!
From level 2... marble does not break!

Strategy for marbles breaking on level 4 and not below:
From level 3... marble does not break!
From level 4... marble breaks!

Strategy for marbles not breaking:
From level 3... marble does not break!
From level 4... marble does not break!
```

[7]: `systematic_two_marbles_simulation(5)`

```
At most 3 drops will be needed.

Strategy for marbles breaking on level 1 and not below:
From level 3... marble breaks!
From level 1... marble breaks!

Strategy for marbles breaking on level 2 and not below:
From level 3... marble breaks!
From level 1... marble does not break!
From level 2... marble breaks!
```

```
Strategy for marbles breaking on level 3 and not below:
From level 3... marble breaks!
From level 1... marble does not break!
From level 2... marble does not break!

Strategy for marbles breaking on level 4 and not below:
From level 3... marble does not break!
From level 5... marble breaks!
From level 4... marble breaks!

Strategy for marbles breaking on level 5 and not below:
From level 3... marble does not break!
From level 5... marble breaks!
From level 4... marble does not break!

Strategy for marbles not breaking:
From level 3... marble does not break!
From level 5... marble does not break!
```

[8]: `systematic_two_marbles_simulation(6)`

```
At most 3 drops will be needed.

Strategy for marbles breaking on level 1 and not below:
From level 3... marble breaks!
From level 1... marble breaks!

Strategy for marbles breaking on level 2 and not below:
From level 3... marble breaks!
From level 1... marble does not break!
From level 2... marble breaks!

Strategy for marbles breaking on level 3 and not below:
From level 3... marble breaks!
From level 1... marble does not break!
From level 2... marble does not break!

Strategy for marbles breaking on level 4 and not below:
From level 3... marble does not break!
From level 5... marble breaks!
From level 4... marble breaks!

Strategy for marbles breaking on level 5 and not below:
From level 3... marble does not break!
From level 5... marble breaks!
From level 4... marble does not break!
```

```
Strategy for marbles breaking on level 6 and not below:
From level 3... marble does not break!
From level 5... marble does not break!
From level 6... marble breaks!

Strategy for marbles not breaking:
From level 3... marble does not break!
From level 5... marble does not break!
From level 6... marble does not break!
```

To simulate the strategy, it is best not to fix the value of `breaking_level`, but instead randomly generate a number between 1 and $n + 1$, that will remain "hidden" and that the simulated strategy will reveal. For this purpose, we use the `randint()` function from the `random` module. In contrast to `randrange()` which, like `range()`, excludes the upper bound provided as second argument, randint() does include the upper bound. The following illustrates, while also demonstrating how the `fromkeys()` method from the `dict` class allows one to create a dictionary from an iterable to generate the keys, with values all set to None by default, which can be changed by providing a second argument to `fromkeys()`, here, 0:

```
[9]: random_numbers = dict.fromkeys(range(1, 5), 0)
     for _ in range(1000):
         random_numbers[randint(1, 4)] += 1

     random_numbers
```

```
[9]: {1: 224, 2: 253, 3: 264, 4: 259}
```

The function defined next is hardly different to `systematic_two_marbles_simulation()`. Besides assigning `breaking_level` a random value, it does not make use of `step` as `d` suffices and can be modified to play the role that `step` plays in `systematic_two_marbles_simulation()`. The output is made more precise, keeping track of the drop numbers, and which one of the first or second marble is used: 明确的

```
[10]: def two_marbles_simulation(n):
          print(f'Experimenting with {n} levels.')
          d = ceil((sqrt(8 * n + 1) - 1) / 2)
          if d == 0:
              print('No drop will be needed.')
          elif d == 1:
              print('At most 1 drop will be needed.\n')
          else:
              print('At most', d, 'drops will be needed.\n')
          print('Now simulating...\n')
          low = 0
          high = n + 1
          drop = 0
          which_marble = 'first'
          breaking_level = randint(1, n + 1)
          while low < high - 1:
              level = min(low + d, high - 1)
```

```
        drop += 1
        if breaking_level <= level:
            print(f'Drop #{drop} with {which_marble} marble, '
                  f'from level {level}... marble breaks!'
                 )
            which_marble = 'second'
            high = level
            d = 1
        else:
            print(f'Drop #{drop} with {which_marble} marble, '
                  f'from level {level}... marble does not break!'
                 )
            low = level
            if d > 1:
                d -= 1
    if high == n + 1:
        print("Tower would have to be higher to reveal marbles's limits.")
    elif high == 1:
        print(f'Marbles break when dropped from the first level.')
    else:
        print(f'Marbles break when dropped from level {high}, not below.')
```

[11]: `two_marbles_simulation(30)`

```
Experimenting with 30 levels.
At most 8 drops will be needed.

Now simulating...

Drop #1 with first marble, from level 8... marble breaks!
Drop #2 with second marble, from level 1... marble does not break!
Drop #3 with second marble, from level 2... marble does not break!
Drop #4 with second marble, from level 3... marble does not break!
Drop #5 with second marble, from level 4... marble does not break!
Drop #6 with second marble, from level 5... marble does not break!
Drop #7 with second marble, from level 6... marble breaks!
Marbles break when dropped from level 6, not below.
```

[12]: `two_marbles_simulation(0)`

```
Experimenting with 0 levels.
No drop will be needed.
Now simulating...

Tower would have to be higher to reveal marbles's limits.
```

Let us design and implement the solution for an arbitrary value of m. We still reverse the problem: given d, what is the maximum number of levels n (the maximum height of the tower)

such that for any marble quality, one is sure to correctly assess the marbles' quality with no more than d drops? Now, n depends on both d and m; write $H(d, m)$ for n. Trivially, if either $d = 0$ or $m = 0$, then $H(d, m) = 0$. Assume that both d and m are strictly positive.

- If the marble breaks, m - 1 marbles remain and no test needs to be done on higher levels.
- If the marble does not break, m marbles remain and no test needs to be done on lower levels.
- In any case, d - 1 drops remain.

This yields: $H(d, m) = H(d - 1, m - 1) + H(d - 1, m) + 1$.

These equations allow one to compute d as the least integer with $H(d, m) >= n$. For the simulation, if *low* is the largest integer for which it is known that marbles can be dropped from level *low* without breaking, d' is the number of drops that remain, and m' is the number of marbles that remain, then the next drop should be from level $low + H(d' - 1, m' - 1) + 1$, and the numbers of drops and marbles that remain will be trivially updated, depending on the outcome of the last drop for the latter, but not for the former.

For all nonnegative integers k and n, set

- $B(0, k) = 1$,
- $B(n, 0) = 1$, and
- $B(n + 1, k + 1) = B(n, k) + B(n, k + 1)$.

These recurrence relations are identical to those that determine the binomial coefficients. Let us verify the following:

1. $H(n, k) = B(n, k) - 1$.
2. If $k > n$ then $B(n, k) = B(n, n)$.
3. If $k <= n$ then $B(n, k) = \Sigma_{k'=0}^{k} \binom{n}{k'}$.

For 1.: The equality is trivial if either $n = 0$ or $k = 0$. Suppose that both $n > 0$ and $k > 0$, and $H(n', k') = B(n', k') - 1$ for all $n' \leq n$ and $k' \leq k$ with either $n' < n$ or $k' < k$. Then $H(n, k)$ is equal to $H(n1, k1) + H(n1, k) + 1$, which is equal to $B(n1, k1) - 1 + B(n1, k) - 1 + 1$, which is equal to $B(n1, k1) + B(n1, k) - 1$, which is equal to $B(n, k) - 1$.

For 2.: The equality is trivial if $n = 0$. Suppose that $n > 0$ and that $B(n', k) = B(n', n')$ for all $n' < n$ and $k > n'$. Then for all $k > n$, $B(n, k)$ is equal to $B(n - 1, k - 1) + B(n - 1, k)$, which is equal to $B(n - 1, n - 1) + B(n - 1, n)$, which is equal to $B(n, n)$.

For 3.: The equality is trivial if $k = 0$. Suppose that $k > 0$, and that the equality holds for any $k' < k$ and $n' \geq k'$. Then for all $n \geq k$, $B(n, k) = B(n - 1, k - 1) + B(n - 1, k) = \Sigma_{k'=0}^{k-1} \binom{n-1}{k'} + \Sigma_{k'=0}^{k} \binom{n-1}{k'} = \Sigma_{k'=1}^{k} [\binom{n-1}{k'-1} + \binom{n-1}{k'}] + \binom{n-1}{0} = \Sigma_{k'=1}^{k} \binom{n}{k'} + \binom{n}{0} = \Sigma_{k'=0}^{k} \binom{n}{k'}$.

The values $B(n, k)$ determine Bernouilli rectangle, whose rows are computed similarly to the rows of Pascal triangle:

1	1	1	1	1	1	1	...
1	2	2	2	2	2	2	...
1	3	4	4	4	4	4	...
1	4	7	8	8	8	8	...
1	5	11	15	16	16	16	...
1	6	16	26	31	32	32	...
1	7	22	42	57	63	64	...
...	...	...	...	...	...	...	...

Let us summarise the key ingredients thanks to which d can be computed:

- $H(d, m) = B(d, m) - 1$
- $B(d, m)$ is equal to the value of Bernouilli triangle at the intersection of row number d and column number m (with both numberings starting from 0).
- $H(d, m)$ is the maximum number of levels a tower can have so that with at most d drops and using at most m marbles, it is possible to know the lowest L, if any, such that a marble dropped from L breaks but does not break when dropped from below L.
- m, the number of marbles, and n, the tower height, are known, and d has to be determined from m and n.

It follows that:

- d is the least integer such that $H(m, d) \geq n$, hence the least integer such that $B(m, d) > n$;
- to compute d, it suffices to build the first m columns of Bernouilli triangle for row 1, row 2... up to the first row whose mth column contains a number greater than n. For instance, if $m = 4$ and $n = 25$, then $d = 5$ since column number 4 of Bernouilli rectangle reads as 1, 2, 4, 8, 16, 31..., with 31, first number in the sequence greater than 25, on row number 5.

Given nonnegative integers p and q, the first p rows of the first q columns of Bernouilli rectangle can be represented as a list of p lists, each of which is a list of q numbers. For instance, the first 3 rows of the first 4 columns of Bernouilli rectangle can be represented as:

```
[13]: [[1, 1, 1, 1],
       [1, 2, 2, 2],
       [1, 3, 4, 4]
      ]
```

```
[13]: [[1, 1, 1, 1], [1, 2, 2, 2], [1, 3, 4, 4]]
```

Before we build the section of Bernouilli rectangle of interest thanks to which we can determine d, and then run the simulation, let us work with a rectangle of size 8 x 10, filled with all integers from 0 up to 79, the integers between 0 and 9 making up the first row, those between 10 and 19 the second row, etc. To this end, we create a list `rectangle` of 8 elements, each of which is a list of 10 elements; each element of `rectangle` represents a row and is conveniently defined with a list comprehension, while `rectangle` itself represents the sequence of all rows and is also conveniently defined with a list comprehension:

```
[14]: rectangle = [[10 * i + j for j in range(10)] for i in range(8)]

      rectangle
```

```
[14]: [[0, 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]]
```

Let us write a function to display the contents of `rectangle` in a way that is consistent with that representation. The largest number, 79, is stored in the bottom right corner: it is the last member of the last member of `rectangle`, hence it can be retrieved as `rectangle[-1][-1]`. It consists

of 2 digits, with 2 equal to `len(str(79))`. The `display()` function is appropriate to display the contents of any rectangle of integers by nicely aligning all values in a given column, under the assumption that the space taken by integers in the rectangle is maximal for the integer in the bottom right corner:

```python
[15]: def display(rectangle):
          field_width = len(str(rectangle[-1][-1]))
          for row in rectangle:
              print(' '.join(f'{e:{field_width}}' for e in row))

      display(rectangle)
```

```
 0  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
```

When defining `rectangle`, we could have started with an empty list *L*, and 8 times, appended *L* with an appropriate list of 10 elements:

```python
[16]: rectangle = []
      for i in range(8):
          rectangle.append([10 * i + j for j in range(10)])

      display(rectangle)
```

```
 0  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
```

We use this technique to build the section of Bernouilli rectangle of interest, starting with a list of one list rather than with an empty list. We trace the construction row by row. The number of rows eventually constructed is 1 more than the maximum of drops:

```python
[17]: def bernouilli_rectangle(m, n):
          bernouilli_rows = [[1] * (m + 1)]
          while bernouilli_rows[-1][m] <= n:
              row = bernouilli_rows[-1]
              bernouilli_rows.append([1] + [row[i - 1] + row[i]
                                              for i in range(1, len(row))
                                            ]
```

```
                                )
        display(bernouilli_rows)
        print()
    return bernouilli_rows

bernouilli_rows = bernouilli_rectangle(4, 25)
print('The maximum number of drops is', len(bernouilli_rows) - 1)
```

```
1 1 1 1 1
1 2 2 2 2

1 1 1 1 1
1 2 2 2 2
1 3 4 4 4

1 1 1 1 1
1 2 2 2 2
1 3 4 4 4
1 4 7 8 8

 1   1   1   1   1
 1   2   2   2   2
 1   3   4   4   4
 1   4   7   8   8
 1   5  11  15  16

 1   1   1   1   1
 1   2   2   2   2
 1   3   4   4   4
 1   4   7   8   8
 1   5  11  15  16
 1   6  16  26  31

The maximum number of drops is 5
```

The simulation is very similar to the 2 marbles case. The key difference is in comput-ing the level from which a marble should then be dropped: first `d` is decremented by 1, and then the assignment `level = min(low + d, high - 1)` is changed to `level = min(low + bernouilli_rows[d][m - 1], high - 1)`. Indeed:

- we noticed that if *low* is the largest integer for which it is known that marbles can be dropped from level *low* without breaking, d' is the number of drops that remain, and m' is the number of marbles that remain, then the next drop should be from level $low + H(d' - 1, m' - 1) + 1$;
- we established that $H(d', m') = B(d', m') - 1$.

The necessary values (and others) and precomputed in `bernouilli_rows`, which allows the rest of the code to be slightly simpler than with the 2 marbles case:

```
[18]: def m_marbles_simulation(m, n):
          print(f'Experimenting with {m} marbles and {n} levels.')
          bernouilli_rows = [[1] * (m + 1)]
          while bernouilli_rows[-1][m] <= n:
              row = bernouilli_rows[-1]
              bernouilli_rows.append([1] + [row[i - 1] + row[i]
                                            for i in range(1, len(row))
                                           ]
                                    )
          d = len(bernouilli_rows) - 1
          if d == 0:
              print('No drop will be needed.')
          elif d == 1:
              print('At most 1 drop will be needed.\n')
          else:
              print('At most', d, 'drops will be needed.\n')
          print('Now simulating...\n')
          low = 0
          high = n + 1
          drop = 0
          which_marble = 1
          breaking_level = randint(1, n + 1)
          while low < high - 1:
              d -= 1
              level = min(low + bernouilli_rows[d][m - 1], high - 1)
              drop += 1
              if breaking_level <= level:
                  print(f'Drop #{drop} with marble #{which_marble}, '
                        f'from level {level}... marble breaks!'
                       )
                  which_marble += 1
                  high = level
                  m -= 1
              else:
                  print(f'Drop #{drop} with marble #{which_marble}, '
                        f'from level {level}... marble does not break!'
                       )
                  low = level
          if high == n + 1:
              print("Tower would have to be higher to reveal marbles's limits.")
          elif high == 1:
              print(f'Marbles break when dropped from the first level.')
          else:
              print(f'Marbles break when dropped from level {high}, not below.')

[19]: m_marbles_simulation(4, 20)
```

Experimenting with 4 marbles and 20 levels.

```
At most 5 drops will be needed.

Now simulating...

Drop #1 with marble #1, from level 15... marble breaks!
Drop #2 with marble #2, from level 7... marble breaks!
Drop #3 with marble #3, from level 3... marble breaks!
Drop #4 with marble #4, from level 1... marble breaks!
Marbles break when dropped from the first level.
```

[20]: `m_marbles_simulation(7, 2000)`

```
Experimenting with 7 marbles and 2000 levels.
At most 12 drops will be needed.

Now simulating...

Drop #1 with marble #1, from level 1486... marble does not break!
Drop #2 with marble #1, from level 2000... marble breaks!
Drop #3 with marble #2, from level 1868... marble breaks!
Drop #4 with marble #3, from level 1649... marble does not break!
Drop #5 with marble #3, from level 1748... marble does not break!
Drop #6 with marble #3, from level 1805... marble does not break!
Drop #7 with marble #3, from level 1836... marble breaks!
Drop #8 with marble #4, from level 1820... marble does not break!
Drop #9 with marble #4, from level 1828... marble breaks!
Drop #10 with marble #5, from level 1824... marble does not break!
Drop #11 with marble #5, from level 1826... marble breaks!
Drop #12 with marble #6, from level 1825... marble breaks!
Marbles break when dropped from level 1825, not below.
```