香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

S243 - Stop the Avalanche!

Vincent Chiu {VCLH}
2024-02-17

Background

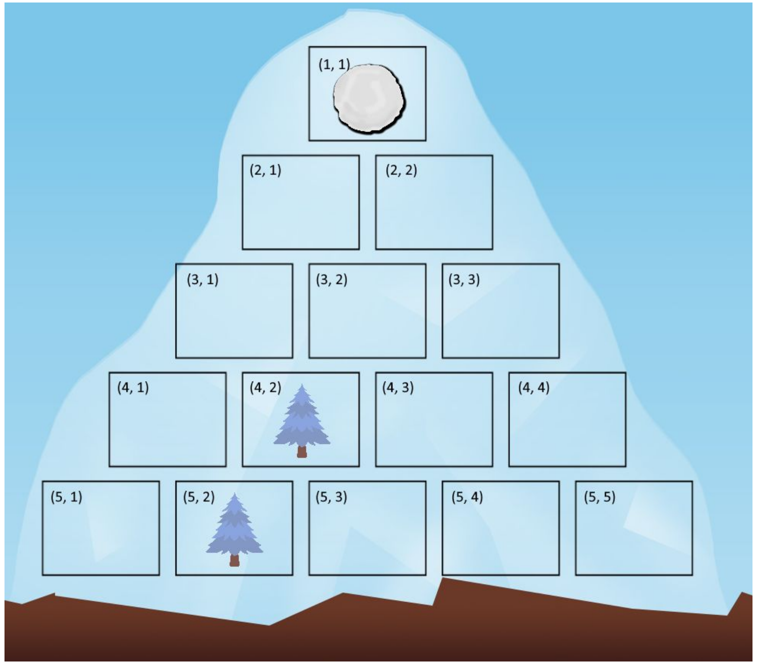
Problem idea by kctung

Preparation by kctung, __declspec and VCLH

Illustration by microtony and __declspec

Slides by __declspec and VCLH

Presentation by VCLH

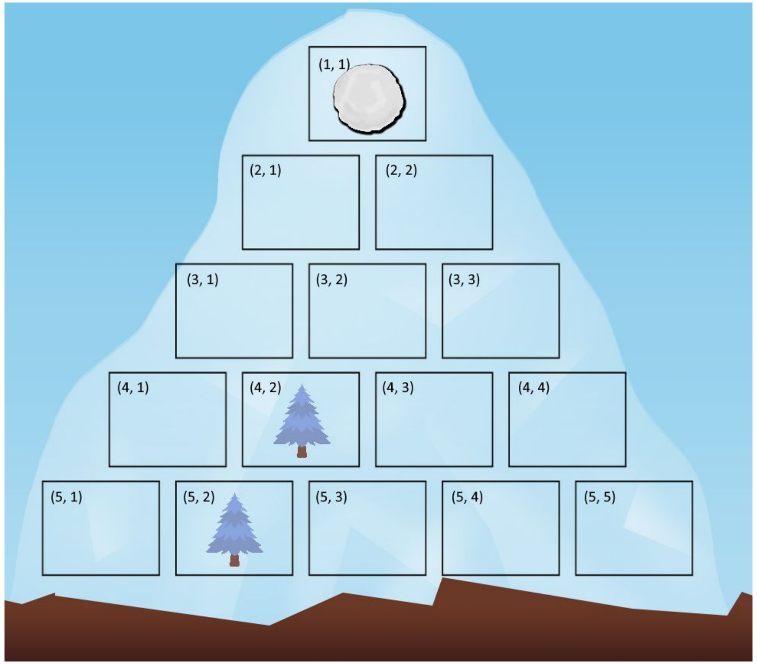


Problem Restatement

In a size-N triangle, some cells are occupied by trees. For each query, a snowball will be rolling down from the top of the hill, whose path is unpredictable.

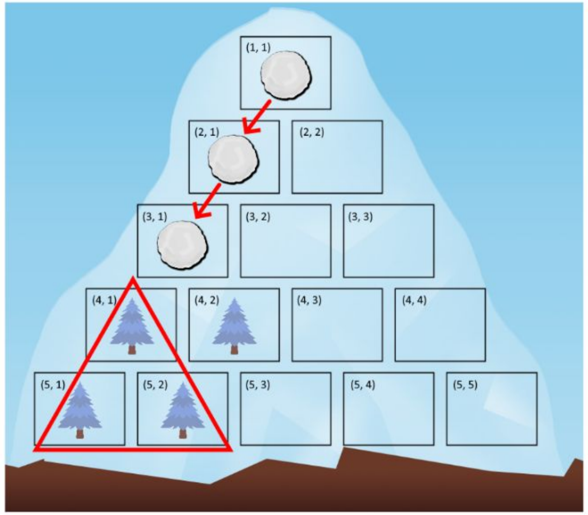
You can fill a size-k' sub-triangle (1 ≤ k' ≤ K[i]) with trees at any moment, depending on the location of the snowball at that moment.

What is the earliest time you can stop the snowball in the worst case in each query?

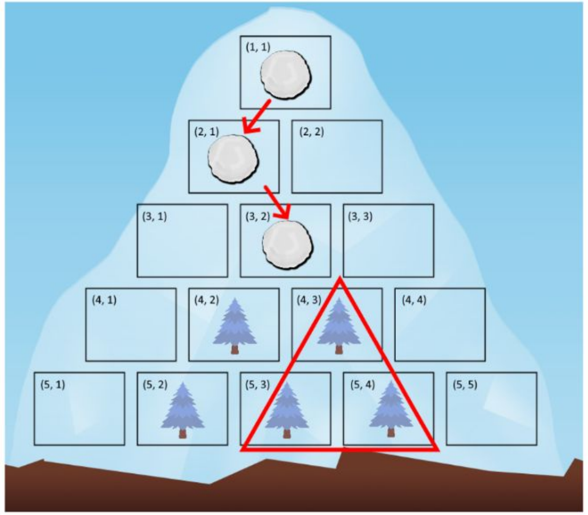


Problem Restatement

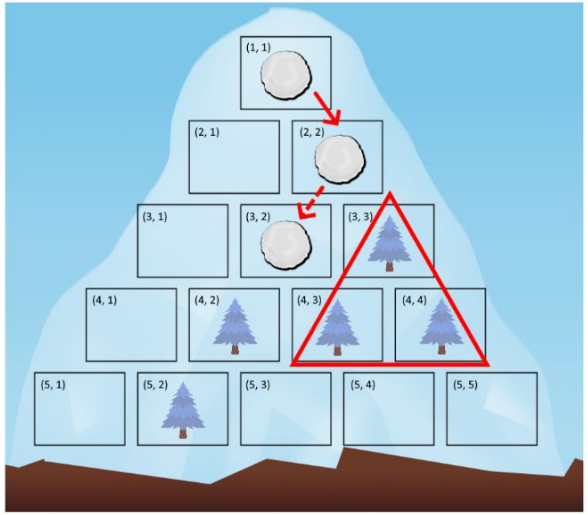
$(1,1) \to (2,1) \to (3,1)$: cast spell $(2,4,1)$ at time $t = 2$, and the snowball stops at $(3,1)$:



$(1,1) \to (2,1) \to (3,2)$: cast spell $(2,4,3)$ at time $t = 2$, and the snowball stops at $(3,2)$:



$(1,1) \to (2,2)$: cast spell $(2,3,3)$ at time $t = 1$, and the snowball stops at $(3,2)$ at time $t = 2$:



Subtasks

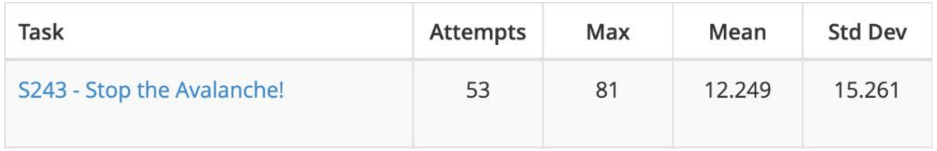
Partial score provided:
For all subtasks, you can score 30% if your program can, in each test case, determine if it's possible to always stop the snowball correctly for every query.

For all cases:
$2 \le N \le 3000$
$1 \le Q \le N$
$T_{i,j} \in \{0, 1\}$ for $1 \le i \le N$ and $1 \le j \le i$
$0 \le K_i \le N - 1$ for $1 \le i \le Q$

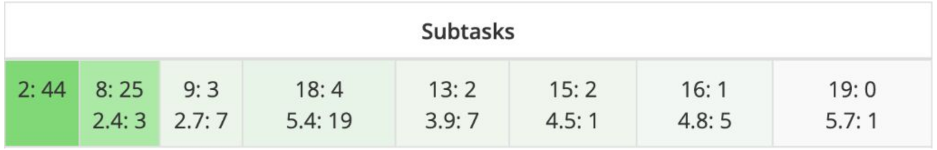
	Points	Constraints
1	2	$N = 2$
2	8	$Q = 1$ $K_1 = 0$
3	9	$Q = 1$ $K_1 = 1$
4	18	$Q = 1$ $N \le 20$
5	13	$Q = 1$ $N \le 300$
6	15	$N \le 300$
7	16	$Q = 1$
8	19	No additional constraints

Statistics

	N	H	B	S	G	T
D.N.A.	17	7	7	1	0	32
0 points	3	0	1	0	0	4
2 points	3	5	5	1	0	14
2.4-7.4 points	1	2	3	1	0	7
8 points	0	0	0	1	0	1
10 points	1	0	1	4	2	8
12.7 points	1	0	0	0	0	1
15.4 points	0	1	2	3	2	8
16.4 points	0	0	0	1	0	1
19.2 points	0	0	1	0	0	1
22 points	0	0	0	1	0	1
23.9 points	0	0	1	0	0	1
26.8 points	1	0	0	1	0	2
37 points	0	0	0	0	1	1
39.4 points	0	0	0	0	1	1
65 points	0	0	0	0	1	1
81 points	0	0	0	0	1	1
Total	27	15	21	14	8	85



Task	Attempts	Max	Mean	Std Dev
S243 - Stop the Avalanche!	53	81	12.249	15.261



Subtasks							
2: 44	8: 25 2.4: 3	9: 3 2.7: 7	18: 4 5.4: 19	13: 2 3.9: 7	15: 2 4.5: 1	16: 1 4.8: 5	19: 0 5.7: 1

No one solved it

Highest score = 81 by WYK19X17

Subtask 1 (2%)

N=2

Sanity check

Exhaustion / Case handling

Subtask 2 (8%)

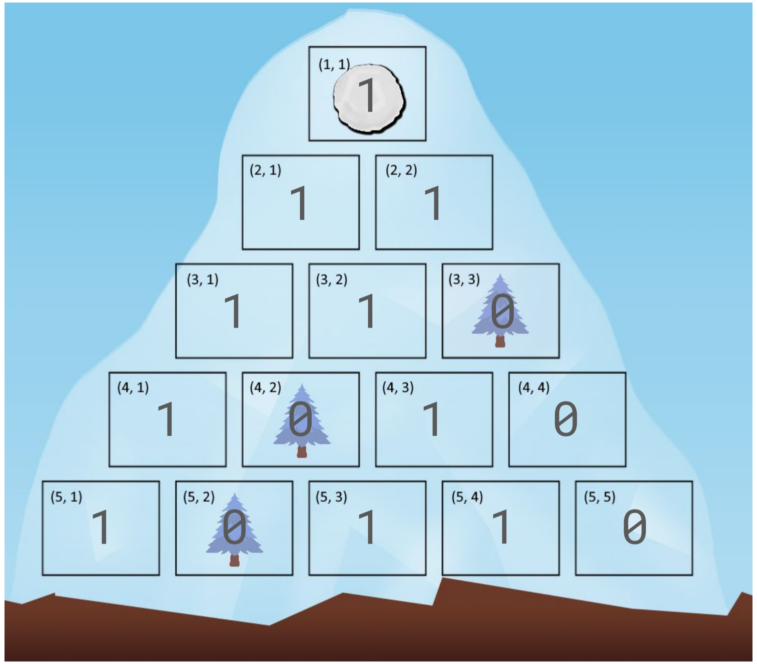
Q=1

K[1]=0

- You cannot cast the spell at all
- Check if the snowball can roll to the bottom by keeping track of the cells it can **reach** from top to bottom
- If it cannot reach any cells in one row, output that row and exit

Reachability

- Perform the algorithm from the top row to the bottom row, like Pascal triangle
- 2D Boolean array: **reachable[i][j]**
  - **reachable[i][j]** is true if snowball can roll from (1,1) to (i,j)
- Simulate how the snowball rolls
  - If (i,j) is not a tree, **reachable[i][j] = reachable[i-1][j] || reachable[i-1][j-1]**



Subtask 3 (9%)

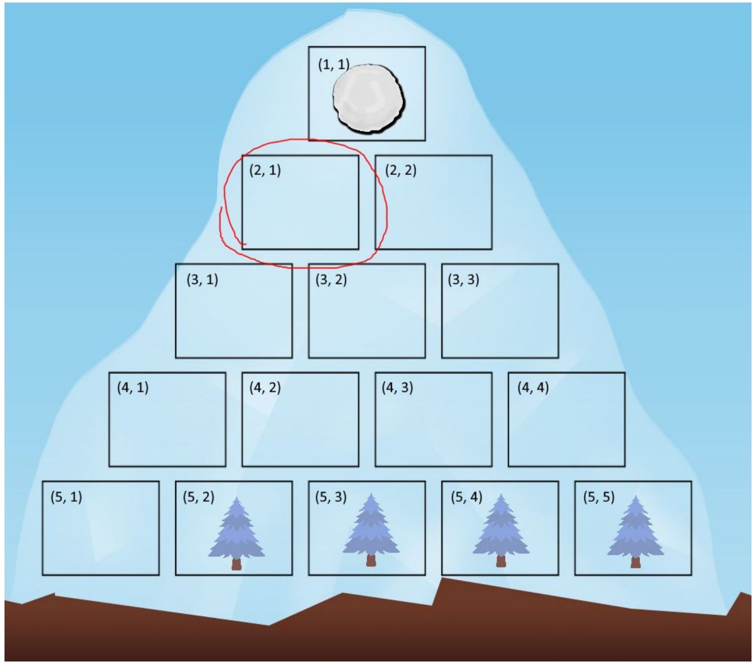
Q=1

K[1]=1

- You must cast the 1-spell directly below the snowball.
- It is always optimal to stop the snowball *immediately* if possible
- If you can cast a 1-spell without stopping the snowball immediately, don’t cast it now; you can always delay it
- Do similar things with **reachability** but snowball can be stopped if any of the two cells below is a tree

Subtask 3 (9%)

- If you cast a 1-spell without stopping the snowball immediately, you can always cast it later
- For example, rather than deploying the spell at (2, 1), we can delay it to (5, 1)

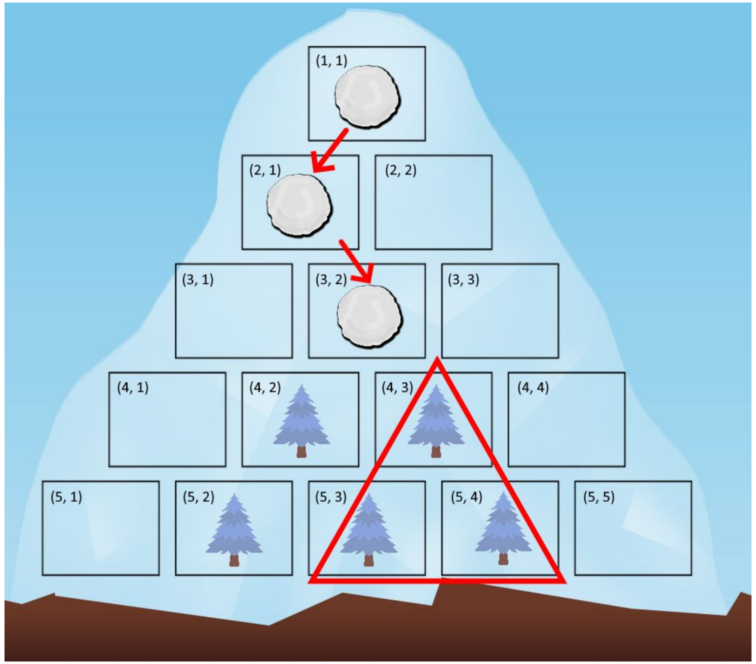


Subtask 4 (18%)

Q=1

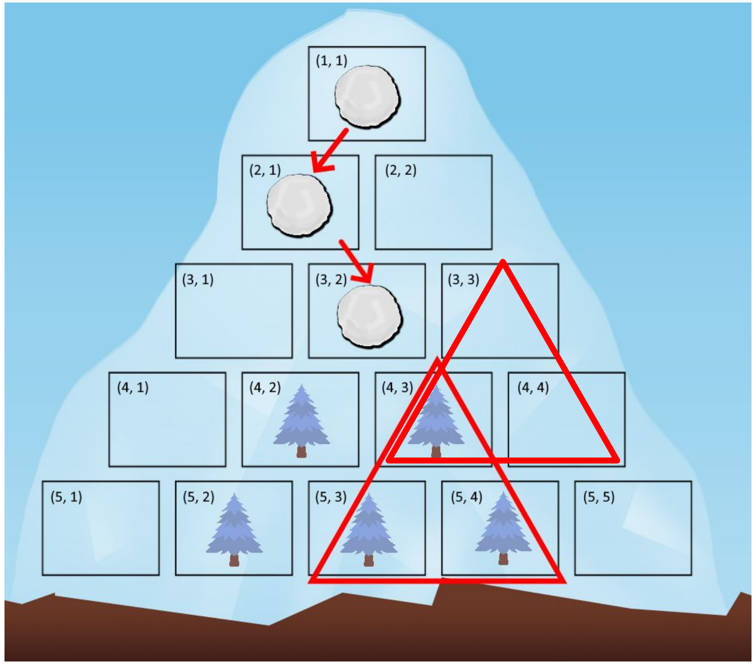
N≤20

- Observation: it is always optimal to cast a spell directly below the snowball (defined by the top of the spell triangle)
- Otherwise, we can wait until the snowball falls to the point where the spell is cast



Subtask 4 (18%)

- Observation: it is always optimal to cast a spell directly below the snowball (defined by the top of the spell triangle)
- For example, at time t=2, rather than deploying the spell at (3, 3), we can deploy it at (4, 3)



Subtask 4 (18%)

- Brute force by DFS
- Simulate the rolling of the snowball at (r, c)
- Try casting the spell with top of triangle at the cells directly bottom of the snowball (if either cell is a tree then we can directly stop the snowball)
  - Calculate the time the snowball stops in the worst case like Subtask 3
- Or not casting the spell at all and let the snowball roll to the cells below
  - The time the snowball stops in the worst case is **max(dfs(r+1,c), dfs(r+1,c+1))**
- See which option stops the snowball the quickest
- One implementation has time complexity: $O(N^4)$, space complexity: $O(N^4)$
  - But constraints are small enough

Subtask 5,6,7 (13%,15%,16%)

Subtask 5: **Q=1, N≤300** | Subtask 6: **N≤300** | Subtask 7: **Q=1**

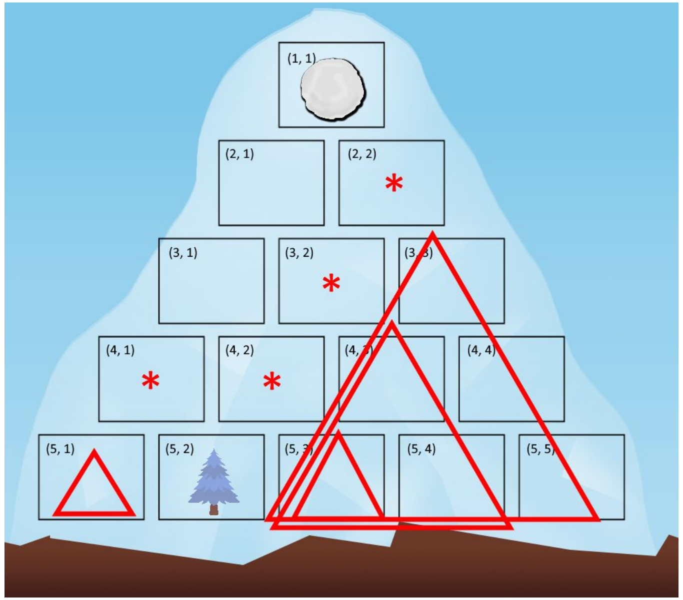
- Repeated calculations of **dfs(r, c)**
- Calculate **bottom-up** like Fibonacci numbers
  - Also known as dynamic programming
- That is, compute value of **dfs(r, c)** by descending row order since the value of **dfs(r, c)** only depend of the value of **dfs(r+1, c)** and **dfs(r+1, c+1)**
- Time complexity: **$O(QN^4)$ or $O(QN^3)$ or $O(QN^2)$** depending how well you optimised the preprocessing
  - We don't need to compute where the snowball stops every time

Full solution

- The previous solutions are still kind of “brute force”
- We need a way to compute the answer for every size of the spell, all at once, in **O(~~Q~~N^2)**
- It is always very tempting to code up suboptimal solutions without proving them, for example “filling gaps” when possible, or assuming it is always optimal to cast the spell at the same level
- In fact, our initial full solution was wrong

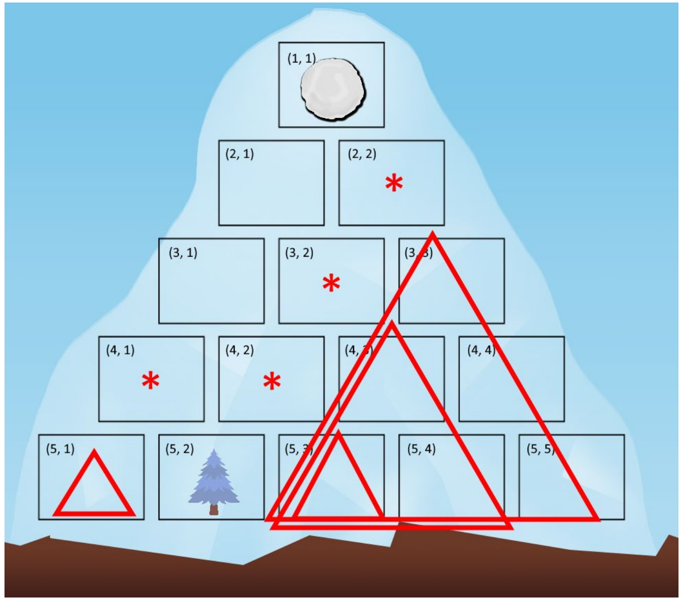
Observation

- To stop the snowball, either it stops at two naturally occurring trees, or a naturally occurring tree and a tree from the spell
- It is always optimal to cast the spell only when the snowball is diagonally above a tree
    - The star positions in the image
    - Each star corresponds to one red triangle which is the spell to be cast to stop it
    - Remember we only cast spells directly under the snowball
- Otherwise just let it roll to a star position and then cast the spell



Observation

- The star positions form the “safety net” of that tree
- Observe that if the snowball is within the safety net, and if the distance between the snowball and the tree is **d**, then a **d**-spell is *sufficient* to stop the snowball (*)
- Particularly, a **d**-spell is only *necessary* **if and only if** the snowball is located on the edge of the net, i.e. on the star positions (**)



Full solution

- Consider this problem: for all cells, if we know the **distance to the closest tree** (below the cell) from there, can we compute the answer?

Consider this configuration:

```
       0
      0 0
     0 0 0
    0 0 0 0
   1 0 0 0 0
  0 0 0 0 1 0
 1 1 1 0 0 0 0
0 0 0 0 1 0 0 1
```

Dist to closest tree from each cell:

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

Full solution

- Let **d[i][j]** be distance to the closest tree below (i, j) (∞ if no trees below)

From (*), we know that

- **d[i][j]** is 1 + the distance to its stopping position if the spell to be cast is determined at time t=(i-1)
- **d[i][j]** is also the sufficient “size” of the spell to be cast at time t=(i-1)

Consider this configuration:

```
              0
            0   0
          0   0   0
        0   0   0   0
      1   0   0   0   0
    0   0   0   0   1   0
  1   1   1   0   0   0   0
0   0   0   0   1   0   0   1
```

d[i][j]:

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

Full solution

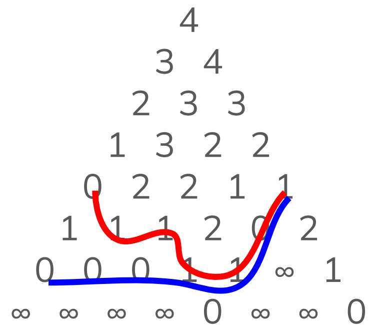
From (**), we know that

- As long as the snowball is still inside the “safety net” of some tree, we can delay the spell till it reaches the edge of the net.
- Delaying means a smaller **d[][]** in the future (lower cells)
- Delaying does not change the stopping position
- ⇒ Earlier d’s are still valid in determining the stopping position, as long as they are within the safety net of the tree

Consider this configuration:

```
       0
      0 0
     0 0 0
    0 0 0 0
   1 0 0 0 0
  0 0 0 0 1 0
 1 1 1 0 0 0 0
0 0 0 0 1 0 0 1
```

d[i][j]:



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

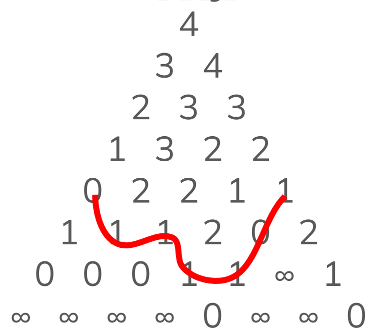
Full solution

- Therefore, we select cells in the triangle using values ≤ **k[i]** (**k[i]**=1 in the example) to form a "tighter" selection
- It means that the snowball can be stopped using a **k[i]**-spell after reaching any cell from that line (by (*))
- Answer = max row of cells + max time in that row - 2 (in this case 7+1-2=6)

Consider this configuration:

```
              0
            0   0
          0   0   0
        0   0   0   0
      1   0   0   0   0
    0   0   0   0   1   0
  1   1   1   0   0   0   0
0   0   0   0   1   0   0   1
```

d[i][j]:



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

Full solution

- Select some cells with values ≤ **k[i]** (=1 on the example) which blocks the snowball from rolling to bottom while **minimising the bottommost row's index**

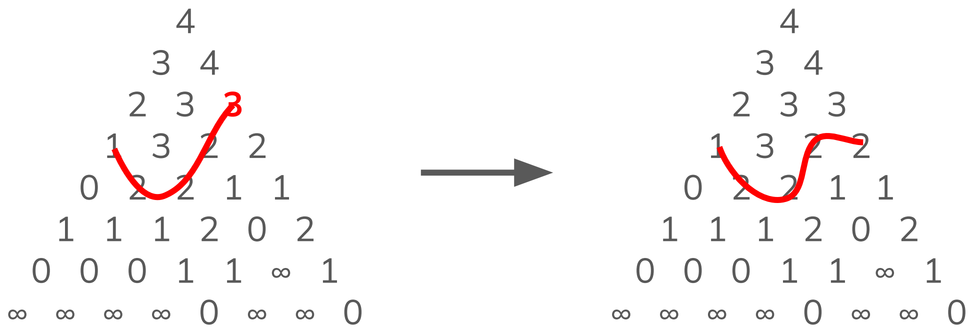
Why is it optimal?

- Consider an optimal selection (bottommost row index minimised, snowball on every cell in the selection can be stopped by **k[i]**-spell), but not necessarily using values ≤ **k[i]**
- Consider value of (i, j) which is > **k[i]**. It can be stopped by a **k[i]**-spell, meaning it is not on the edge of the tree contributing its value (by (**)). Hence, we are able to replace (i, j) with (i+1, j) and (i+1, j+1) and that still works. Note that this process **does not affect the answer**, even if it changes the bottommost row's index. (row+1 but time-1)

Full solution

Consider using a 2-spell. 3 on (3, 3) is not the edge of a safety net, so we can choose the bottom two cells instead.

(Note: the selection is not necessarily a line as we will see.)



Up-propagation

How to construct the distance to tree values?

- Initially every **d[i][j]=∞**
- Calculate **d[i][j]** in descending row order
    - If (i, j) is a tree then **d[i][j]=0**
    - Otherwise **d[i][j]=min(d[i+1][j], d[i+1][j+1])+1**

Consider this configuration:

```
              0
            0   0
          0   0   0
        0   0   0   0
      1   0   0   0   0
    0   0   0   0   1   0
  1   1   1   0   0   0   0
0   0   0   0   1   0   0   1
```

d[i][j]:

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

How to select?

- For each query **q**, let **c[i][j]=1** if **d[i][j] ≤ q** else **0**
- Run reachability (subtask 2) again!
- Time complexity: **$O(QN^2)$** (subtask 7)

Consider this configuration:

```
              0
            0   0
          0   0   0
        0   0   0   0
      1   0   0   0   0
    0   0   0   0   1   0
  1   1   1   0   0   0   0
0   0   0   0   1   0   0   1
```

c[i][j] when q=1:

```
              0
            0   0
          0   0   0
        1   0   0   0
      1   0   0   1   1
    1   1   1   0   1   0
  1   1   1   1   1   0   1
0   0   0   0   1   0   0   1
```

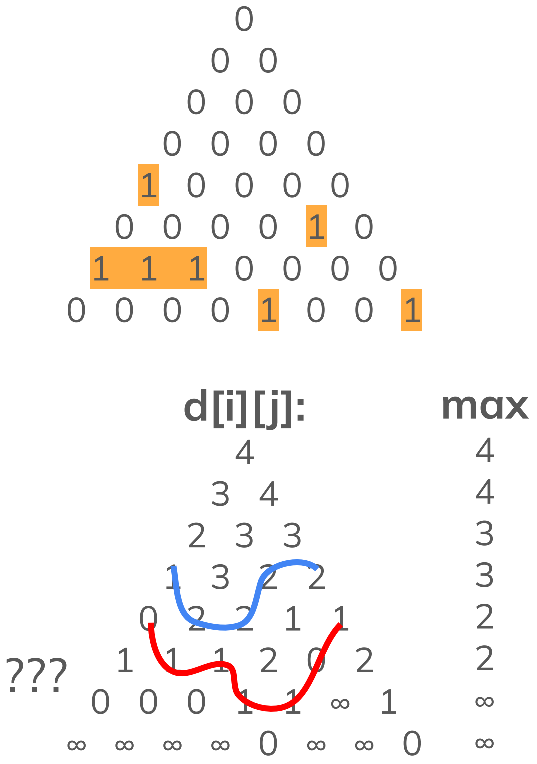
d[i][j]:

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

Is that enough?

- How to select faster?
- ~~Min cut max flow~~
- Let query value be **q**
- Find the first row such that every **d[i][j]** value is ≤ **q**?
- Works for **q=2**, but not quite for **q=1**

Consider this configuration:



```
              0
            0   0
          0   0   0
        0   0   0   0
      1   0   0   0   0
    0   0   0   0   1   0
  1   1   1   0   0   0   0
0   0   0   0   1   0   0   1
```

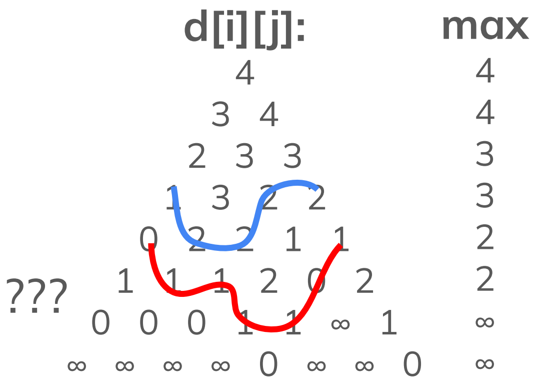
```
            d[i][j]:                 max
               4                      4
             3   4                    4
           2   3   3                  3
         1   3   2   2                3
       0   2   2   1   1              2
     1   1   1   2   0   2            2
???  0   0   0   1   1   ∞   1        ∞
   ∞   ∞   ∞   ∞   0   ∞   ∞   0      ∞
```

Is that enough?

- How to select?
- Consider **q=1,** the red line
- If we want to use the row-maximum strategy, we don't want the values below the red selection to increase row maxes below
- We can basically limit everything below the red line with ≤ q

Consider this configuration:

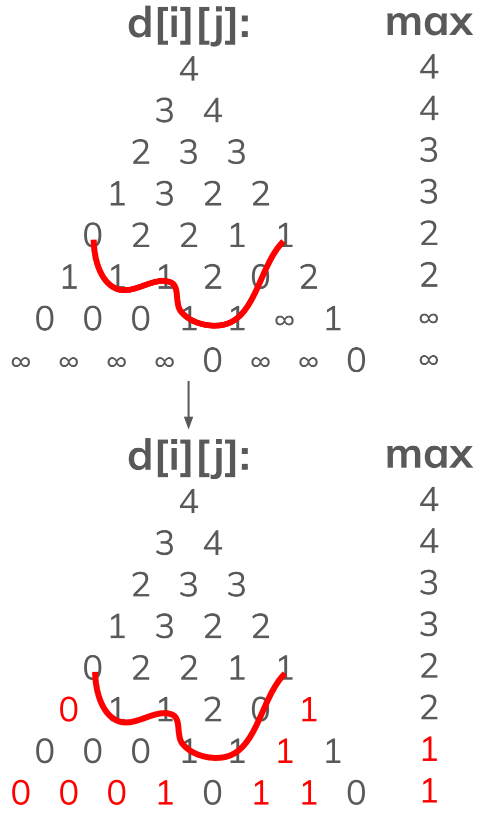
```
              0
            0   0
          0   0   0
        0   0   0   0
      1   0   0   0   0
    0   0   0   0   1   0
  1   1   1   0   0   0   0
0   0   0   0   1   0   0   1
```



```
           d[i][j]:                  max
              4                       4
            3   4                     4
          2   3   3                   3
        1   3   2   2                 3
      0   2   2   1   1               2
???  1   1   1   2   0   2            2
  0   0   0   1   1   ∞   1           ∞
∞   ∞   ∞   ∞   0   ∞   ∞   0         ∞
```

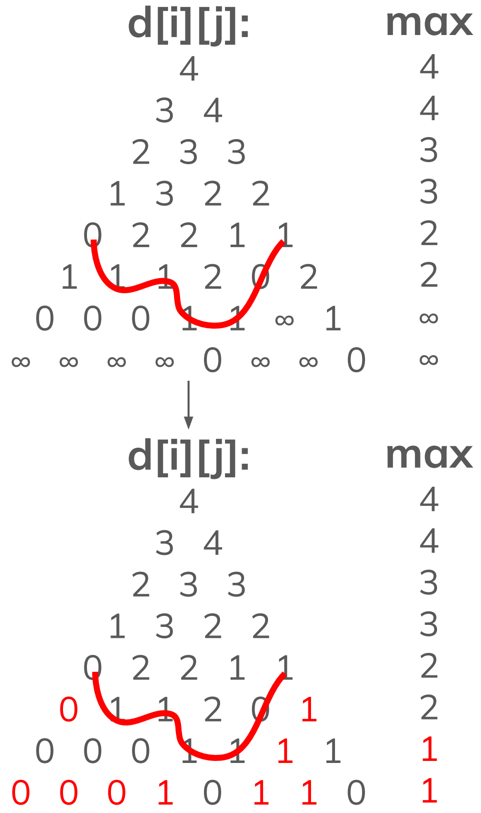
Is that enough?

- How to select?
- Consider **q=1,** the red line
- If we want to use the row-maximum strategy, we don't want the values below the red selection to increase row maxes below
- We can basically limit everything below the red line with ≤ q
- Lemma: this doesn't hinder the calculations of other queries (***)



Lemma (***)

- Lemma: this doesn't hinder the calculations of other queries
- Suppose we are updating with respect to query **q**'s optimal selection
- For all queries **p>q**, the lower bound of the answer is given **q**'s selection and as things above the selection don't change, their answers are not affected
- For all queries **p<q**, the cells **d[i][j]>p** remains the same since only limit to **q**



Down-propagation

- Here comes the magic: how do you perform the operation for every optimal line for every query?
- Update (i, j) from top-down:
    - Let **m=max(d[i-1][j], d[i-1][j-1])**
    - Update **d[i][j]** to **min(d[i][j], m)**
- Every possible selection gets pushed downwards

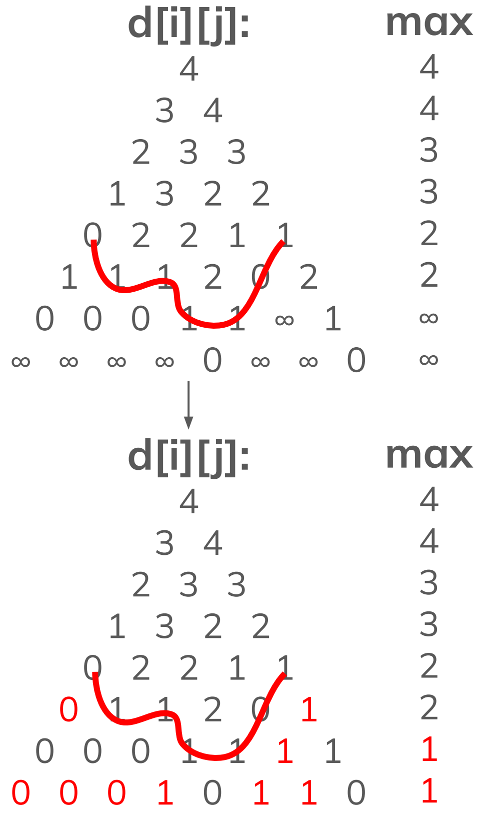
Down-propagation

Originally,

- **d[i][j]** is 1 + the distance to its stopping position if the spell to be cast is determined at time t=(i-1) **at time t=(i-1)**, as well as the sufficient "size" of the spell

After down-propagation,

- **d[i][j]** = minimum size of spell to be cast **at or before time t=(i-1)** for the snowball to stop on or before row **(i+d[i][j])**

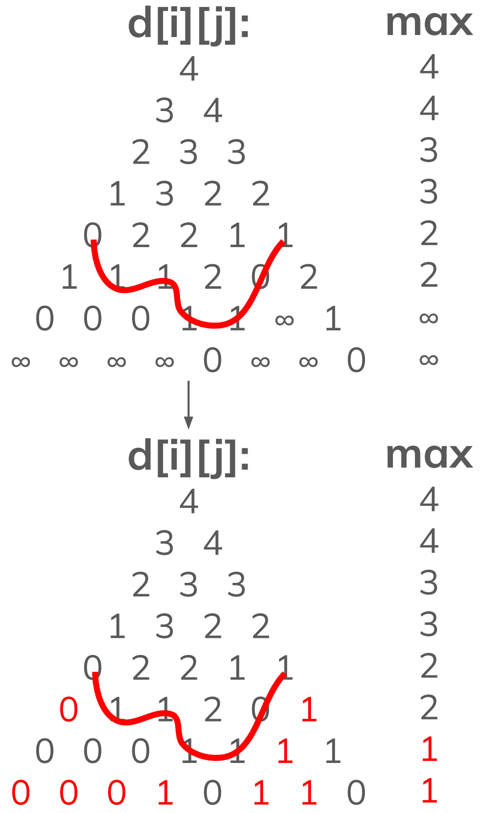


Row summary

- And then the row maximum works now!
- Compute **$r[i] = \max_j\{d[i][j]\}$** after the up- and down-propagation
- For each query **q**, find the smallest row index **i** such that **$r[i] \le q$**. Then the answer would be **$r[i]+i-2$** (why -2?)

Small example

- Query 0: no selection can be made (all **r[i]>0**) so output -1
- Query 1: output 7+1-2=6
- Query 2: output 5+2-2=5
- Query 3: output 3+3-2=4
- Query 8: output 1+4-2=3



Large example

tree[i][j]:

```
                                    0
                                  0   0
                                0   0   0
                              0   0   0   0
                            0   0   0   0   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                          0   0   0   0   0   0
                        0   0   0   0   0   0   0
                      0   0   0   0   0   0   0   0
                    0   0   0   0   0   0   0   0   0
                  0   0   0   0   0   0   0   0   0   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                0   1   1   1   0   0   0   1   1   1   0
              0   0   0   0   0   0   0   0   0   0   0   0
            0   0   0   0   0   0   0   0   0   0   0   0   0
          0   0   0   0   0   0   0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      0   0   0   0   0   1   1   0   0   1   1   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
0   0   0   0   0   0   0   0   1   0   1   0   0   0   0   0   0   0   0
```

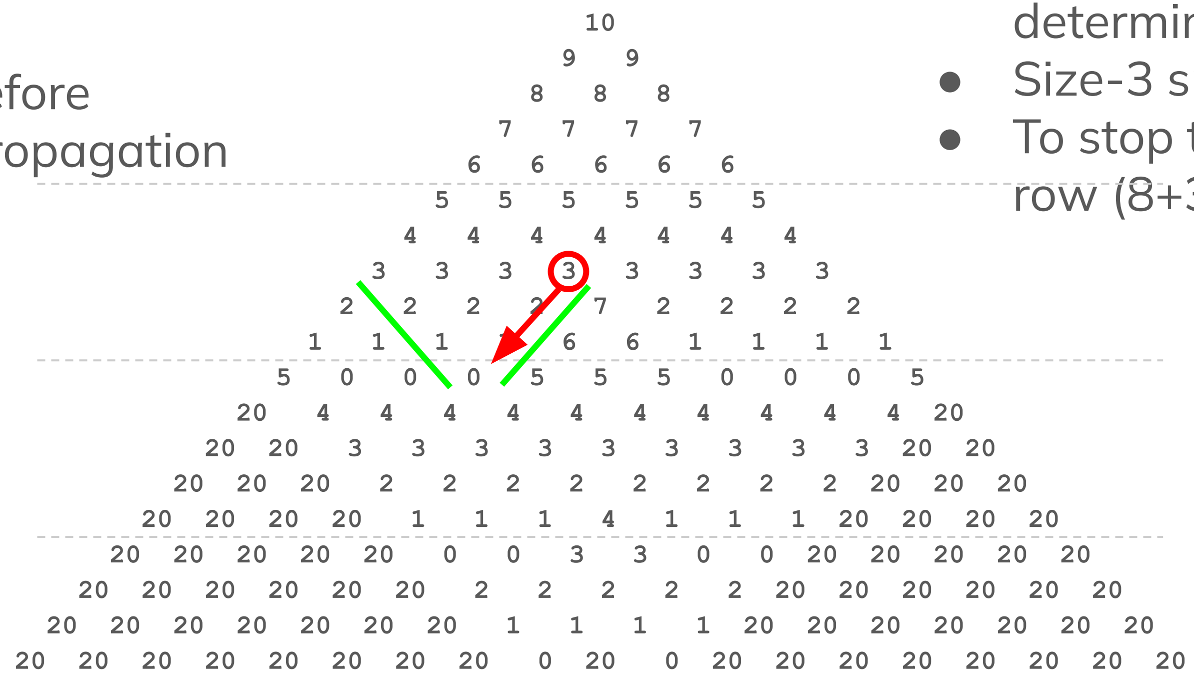
Large example

d[i][j] before down-propagation

```
                                      10
                                     9   9
                                   8   8   8
                                 7   7   7   7
                               6   6   6   6   6
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                             5   5   5   5   5   5
                           4   4   4   4   4   4   4
                         3   3   3   3   3   3   3   3
                       2   2   2   2   7   2   2   2   2
                     1   1   1   1   6   6   1   1   1   1
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                   5   0   0   0   5   5   5   0   0   0   5
                20   4   4   4   4   4   4   4   4   4   4  20
              20  20   3   3   3   3   3   3   3   3   3  20  20
            20  20  20   2   2   2   2   2   2   2   2  20  20  20
          20  20  20  20   1   1   1   4   1   1   1  20  20  20  20
  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
        20  20  20  20  20   0   0   3   3   0   0  20  20  20  20  20
      20  20  20  20  20  20   2   2   2   2   2  20  20  20  20  20  20
    20  20  20  20  20  20  20   1   1   1   1  20  20  20  20  20  20  20
  20  20  20  20  20  20  20  20   0  20   0  20  20  20  20  20  20  20  20
```

Large example

d[i][j] before down-propagation



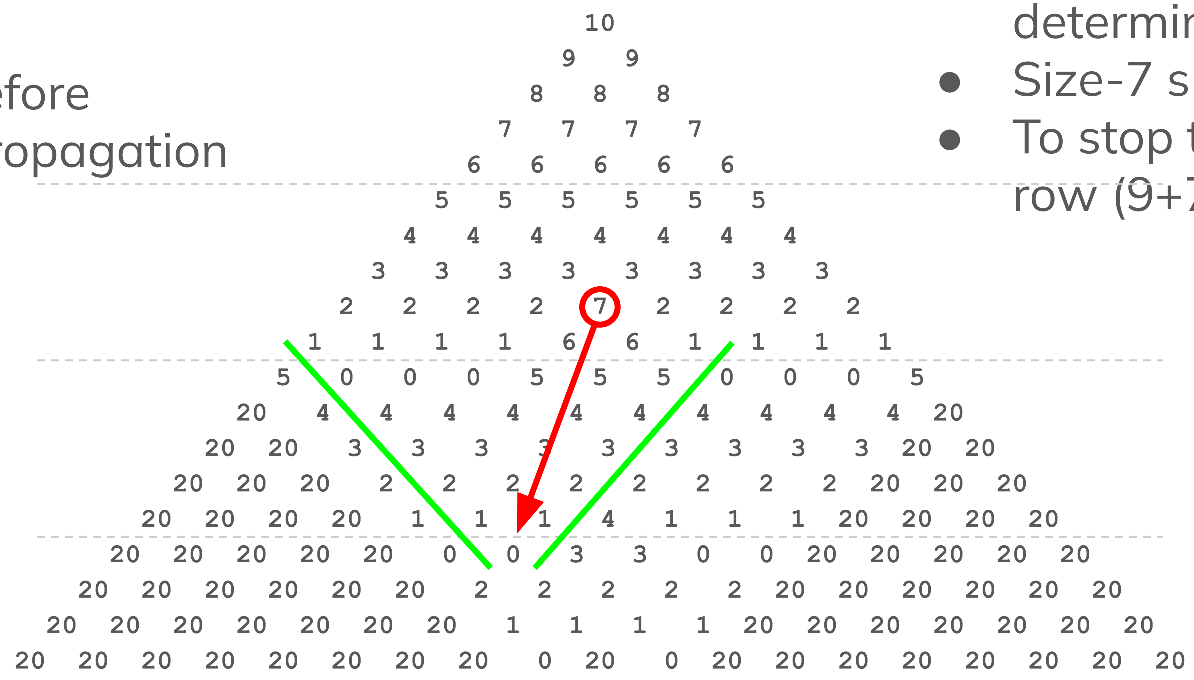
```
                                      10
                                     9   9
                                   8   8   8
                                 7   7   7   7
                               6   6   6   6   6
                             5   5   5   5   5   5
                           4   4   4   4   4   4   4
                         3   3   3   3   3   3   3   3
                       2   2   2   2   7   2   2   2   2
                     1   1   1   1   6   6   1   1   1   1
                   5   0   0   0   5   5   5   0   0   0   5
                20   4   4   4   4   4   4   4   4   4   4  20
              20  20   3   3   3   3   3   3   3   3   3  20  20
            20  20  20   2   2   2   2   2   2   2   2  20  20  20
          20  20  20  20   1   1   1   4   1   1   1  20  20  20  20
        20  20  20  20  20   0   0   3   3   0   0  20  20  20  20  20
      20  20  20  20  20  20   2   2   2   2   2  20  20  20  20  20  20
    20  20  20  20  20  20  20   1   1   1   1  20  20  20  20  20  20  20
  20  20  20  20  20  20  20  20   0  20   0  20  20  20  20  20  20  20  20
```

d[8][4] = 3:

- At time t=7, it is determined that
- Size-3 spell is sufficient
- To stop the snowball at row (8+3-1) = 10

Large example

d[i][j] before down-propagation



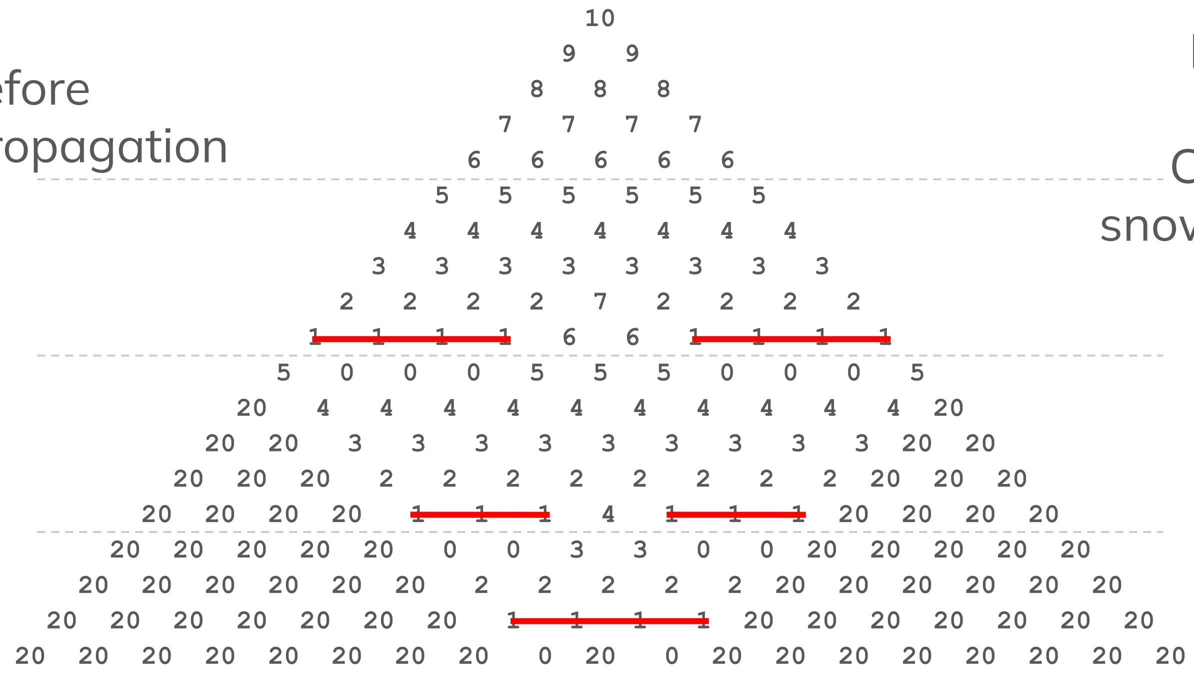
d[9][5] = 7:

- At time t=8, it is determined that
- Size-7 spell is sufficient
- To stop the snowball at row (9+7-1) = 15

Large example

d[i][j] before down-propagation

q=1



```
                                    10
                                  9    9
                               8    8    8
                            7    7    7    7
                         6    6    6    6    6
                      5    5    5    5    5    5
                   4    4    4    4    4    4    4
                3    3    3    3    3    3    3    3
             2    2    2    2    7    2    2    2    2
          1    1    1    1    6    6    1    1    1    1
       5    0    0    0    5    5    5    0    0    0    5
   20    4    4    4    4    4    4    4    4    4    4   20
  20   20    3    3    3    3    3    3    3    3    3   20   20
 20   20   20    2    2    2    2    2    2    2    2   20   20   20
20   20   20   20    1    1    1    4    1    1    1   20   20   20   20
20   20   20   20   20    0    0    3    3    0    0   20   20   20   20   20
20   20   20   20   20   20    2    2    2    2    2   20   20   20   20   20   20
20   20   20   20   20   20   20    1    1    1    1   20   20   20   20   20   20   20
20   20   20   20   20   20   20   20    0   20    0   20   20   20   20   20   20   20   20
```

Red=selection

Can check that snowball can't fall through

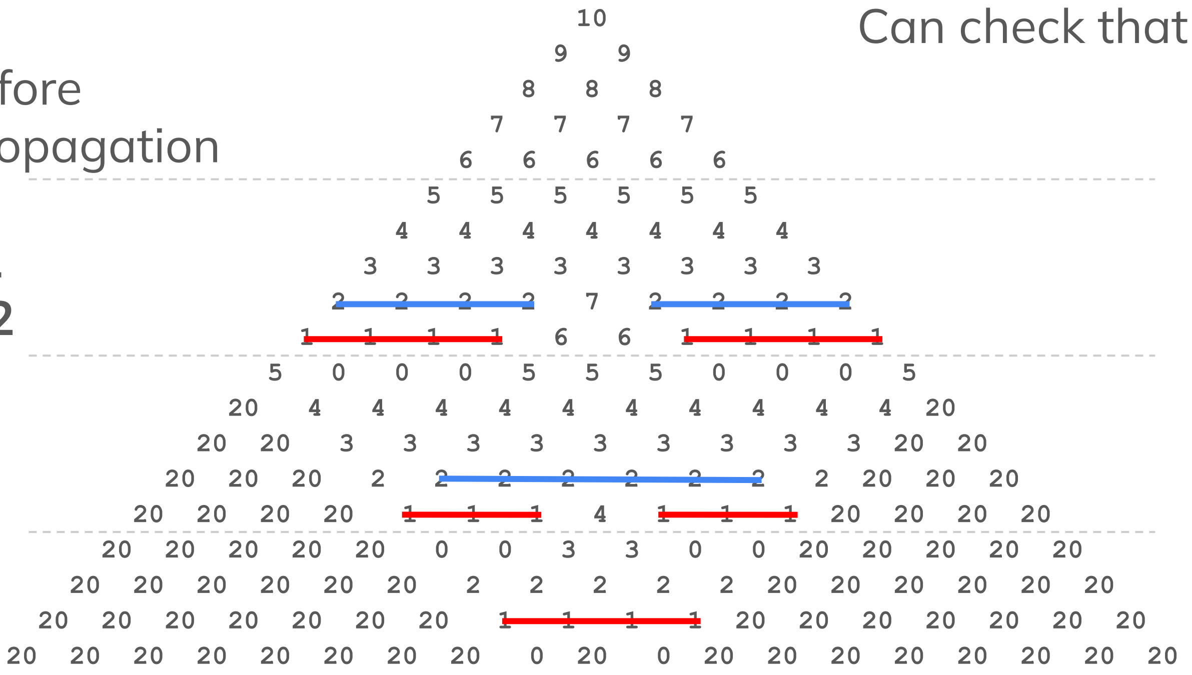
Large example

Numbers on lines = selected

Can check that snowball can't fall through

d[i][j] before down-propagation

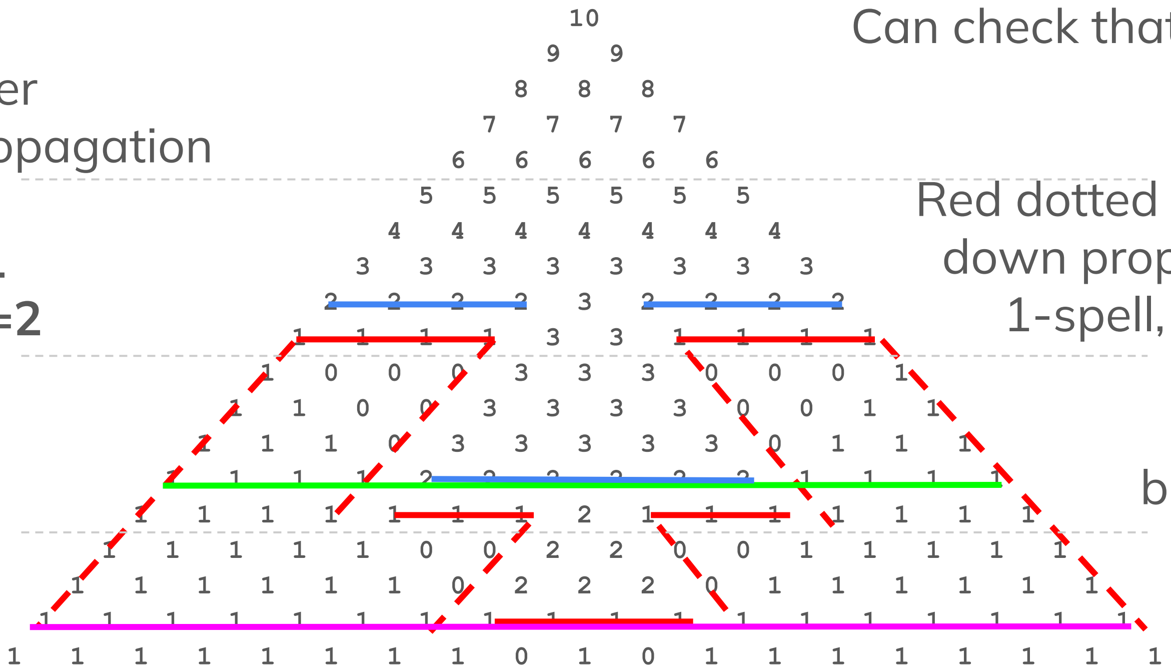
Red: q=1
Blue: q=2



Large example

d[i][j] after down-propagation

Pink: q=1
Green: q=2



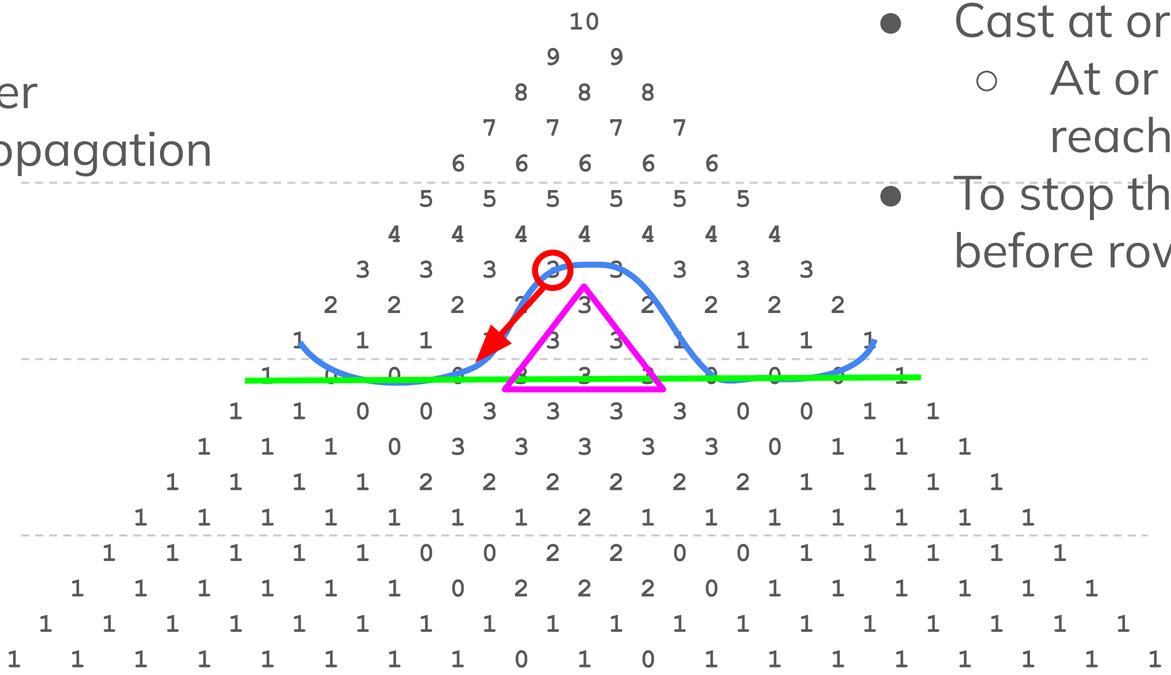
Numbers on lines = selected

Can check that snowball can't fall through

Red dotted lines simulating down propagation: with a 1-spell, snowball won't fall to the cells in the region bounded by lines

Large example

d[i][j] after down-propagation

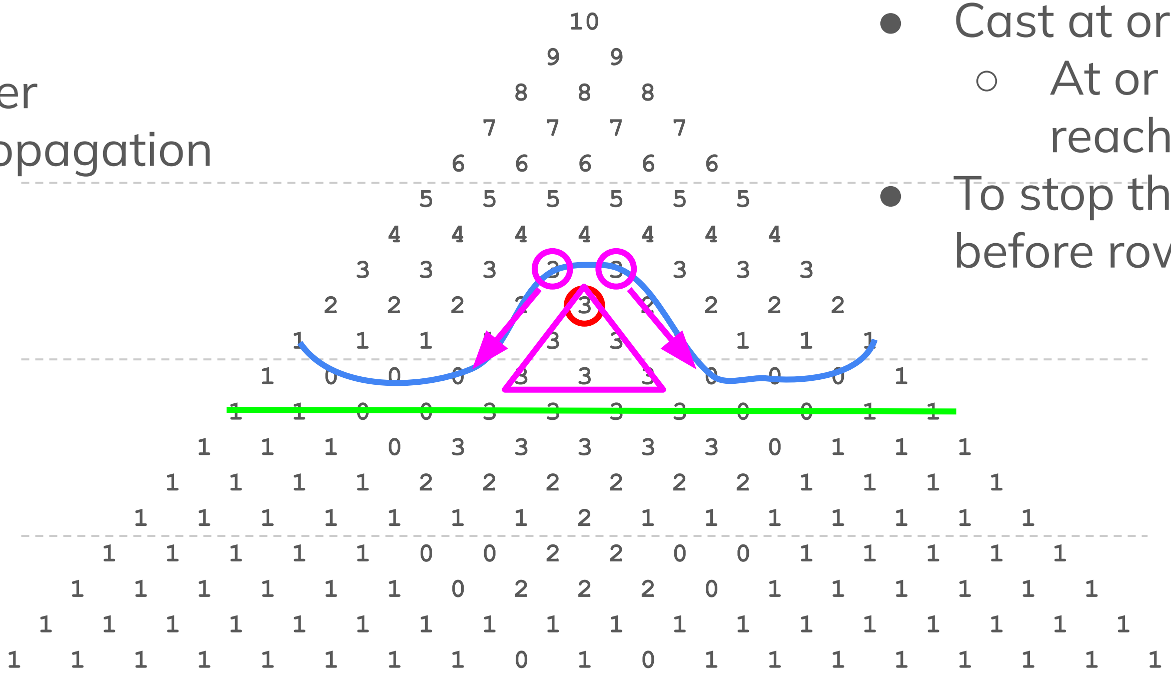


d[8][4] = 3:

- Size-3 spell is enough
- Cast at or before time t=7
    - At or before snowball reaches (8, 4)
- To stop the snowball at or before row (8+3-1) = 10

Large example

d[i][j] after down-propagation



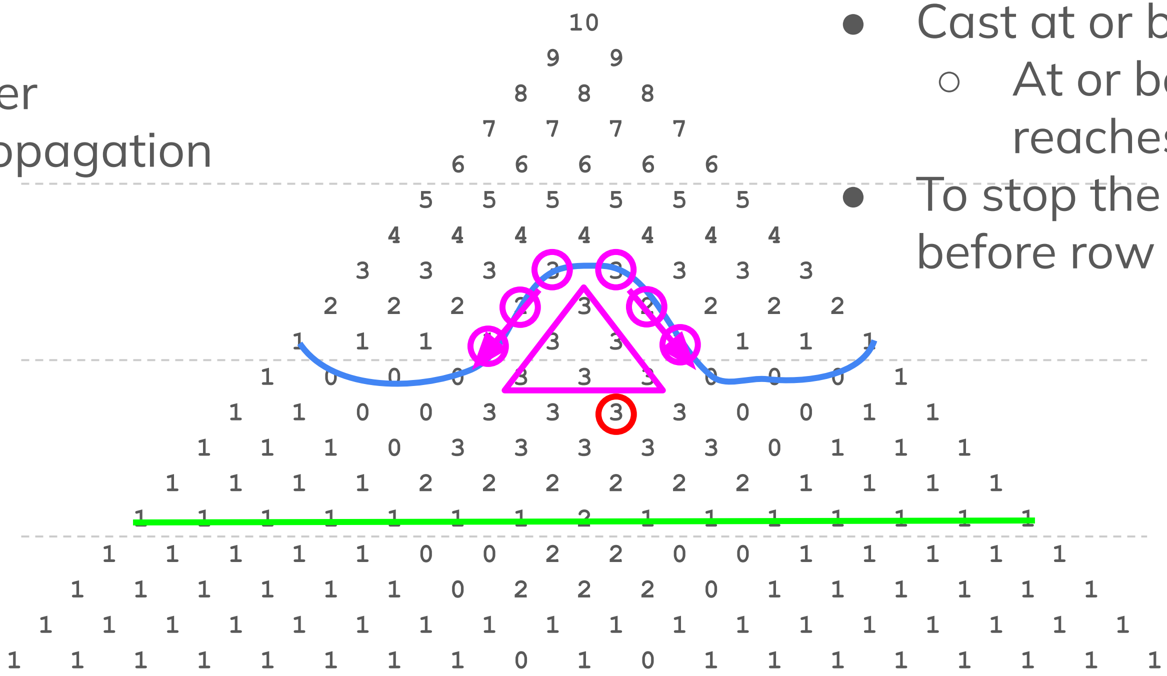
```
                  10
                 9  9
                8  8  8
               7  7  7  7
              6  6  6  6  6
             5  5  5  5  5  5
            4  4  4  4  4  4  4
           3  3  3  3  3  3  3  3
          2  2  2  2  3  2  2  2  2
         1  1  1  1  3  3  1  1  1  1
        1  0  0  0  3  3  3  0  0  0  1
       1  1  0  0  3  3  3  3  0  0  1  1
      1  1  1  0  3  3  3  3  3  0  1  1  1
     1  1  1  1  2  2  2  2  2  2  1  1  1  1
    1  1  1  1  1  1  1  2  1  1  1  1  1  1  1
   1  1  1  1  1  0  0  2  2  0  0  1  1  1  1  1
  1  1  1  1  1  1  0  2  2  2  0  1  1  1  1  1  1
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
1  1  1  1  1  1  1  1  0  1  0  1  1  1  1  1  1  1  1
```

d[9][5] = 3:

- Size-3 spell is enough
- Cast at or before time t=8
    - At or before snowball reaches (9, 5)
- To stop the snowball at or before row (9+3-1) = 11

Large example

d[i][j] after down-propagation



```
                                    10
                                  9    9
                               8    8    8
                            7    7    7    7
                         6    6    6    6    6
                      5    5    5    5    5    5
                   4    4    4    4    4    4    4
                3    3    3    3    3    3    3    3
             2    2    2    2    3    2    2    2    2
          1    1    1    1    3    3    1    1    1    1
       1    0    0    0    3    3    3    0    0    0    1
     1    1    0    0    3    3    3    3    0    0    1    1
   1    1    1    0    3    3    3    3    3    0    1    1    1
 1    1    1    1    2    2    2    2    2    2    1    1    1    1
1   1    1    1    1    1    1    2    1    1    1    1    1    1    1
1   1    1    1    1    0    0    2    2    0    0    1    1    1    1    1
1   1    1    1    1    1    0    2    2    2    0    1    1    1    1    1    1
1   1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1    1
1   1    1    1    1    1    1    1    0    1    0    1    1    1    1    1    1    1    1
```

d[12][7] = 3:

- Size-3 spell is enough
- Cast at or before time t=11
  - At or before snowball reaches (12, 7)
- To stop the snowball at or before row (12+3-1) = 14

Conclusion

- Up-propagation
- Down-propagation
- Row summary
- Easy 100 points