Code Jam 2014 - Round 1C

Analysis: Enclosure

The goal is to place as few stones as possible to enclose at least **K** intersection points in an **N** * **M** grid. Intuitively, it is wasteful to create more than one enclosure since we can always combine the stones into one larger enclosure that cover same or more intersection points. Thus, we can restrict our search to solutions with one enclosure only.

The figure below shows an enclosure for an **N** = 5 by **M** = 5 grid that covers **K** = 19 intersection points with the minimum number of 11 stones.

```
-***-   # the first row
*XXX*   # intermediate row
*XXX*   # intermediate row
*XX*-   # intermediate row
-**--   # the last row
```

We use `'-'` to represent empty intersection, `'X'` to represent an enclosed intersection point, and `'*'` to represent a stone placed in the intersection point (which is also enclosed). Observe that:

- The stones in the first row and the last row are filled in consecutive position in the row.
- Each intermediate row (i.e., the row except the first and the last row) has exactly two stones (i.e., the left stone and the right stone).

Each intermediate row always contains two stones. If there exists an intermediate row that contains only one stone, then it may form two enclosures that touches their boundaries at that row. Since we are not interested in searching solutions with more than one enclosure, we can restrict each intermediate row to always contains exactly two stones.

What about having more than two stones in an intermediate row? This is wasteful and we can avoid that too. To see this, letâ€™s enumerate all possible ways to move the left stone boundary for the next row and at the same time it explains why each intermediate row always contains two stone in the final configuration. The following figure shows all three possible ways to move the left boundary stone:

```
previous row:  --*XXX     --*XXX      --*XXX
    next row:  -*XXXX     --*XXX      ---*XX
               (expand)  (unchanged)  (shrink)
```

Notice that we can only expand the boundary by one otherwise it will create a gap and it will no longer form a **closed** enclosure. Could we place stones in between to close the gap? Such as (shown in bold):

```
previous row:  --*XXX      -**XXX
    next row:  ***XXXX      *XXXXX
                           (push up)
```

Yes, we can, but it is wasteful because we can always â€œpush upâ€ the placed stone to get one more enclosed intersection point as shown in the right figure. Moreover, we can do another push up on the previous row to the previous-previous row and so on until it is pushed up to the top row (each push up gains one more enclosed intersection point). Thus, in the end, after all the push ups, each intermediate row will contain exactly two stones.

To make the search simple, we do not want any push up to happen. In the search, we fix the number of stones at the top row, and generate the next row and we do not want the next row to alter the previous row ("push up" alters the previous row). This explains why we do not want to expand the left boundary more than one position (i.e., it is simpler just to change the number of stones in the top row and do a separate search).

For the shrink case, observe that we only need to shrink at most one position because shrinking more than one is wasteful since we will need more stones to fill the gaps (to maintain enclosure). See the following examples:

```
previous row:   --*XXX     --***XX
    next row:   ---***X     ----*X
                (case 1)   (case 2)
```

In the shrinking case 1, the bolded stone `'*'` (the rightmost stone) is not needed since it is already enclosed. The shrinking case 2 is wasteful since the bolded stone `'*'` (the middle stone) can be pushed down to enclose one more intersection point (which will still be wasteful because it will then became case 1).

Looking at all the possible ways to move the left boundary, we can conclude that it is sufficient to place exactly two stones in each intermediate row.

For the last row, we close the enclosure by connecting the left and right stone boundary of the previous row by placing the stones consecutively. For example:

```
previous row:  --*XXXXX*--
    last row:  ---******---
```

Note that we only close the enclosure if we are sure that the intersection points enclosed by all the previous rows and the last row is at least **K**.

In summary, our search algorithm goes as follows:

- First place a number of stones consecutively from left to right on the top row
- Place two stones for the following (intermediate) rows by expanding / shrinking / unchanged the left and right boundary with respect to the previous row
- Finally if we have enclosed enough intersection points, close the enclosure by placing stones consecutively at the last row.

We will discuss two common ways to solve this problem. The first is a dynamic programming (DP) solution and the second is a greedy solution.

Dynamic Programming (DP)

Since we brute-force the first row, we only need to perform DP for the intermediate rows and the last row. For each row, we need to know:

- The remaining rows left.
- The left stone boundary position.
- The right stone boundary position.
- The remaining intersection points to enclose.

To minimize the left / right stone boundary position, we transpose the grid (if necessary) so that we ended up with the grid with smaller columns than its rows, without affecting optimality. With this transformation, now the left / right boundary position is at most *sqrt*(**N** * **M**). This leads to O(**N** * *sqrt*(**N** * **M**) * *sqrt*(**N** * **M**) * **N** * **M**) solution. While the amortized cost can be less, it still seems large and has potential to run more than the time limit. Can we do better?

It turns out that the exact position of the left and right stone boundary does not really matter as long as the distance between them is not larger than the column size. This also applies for the stones at the top row. What matters is the number of stones placed consecutively in the top row. Where the stones are exactly placed does not matter as long as the number of stones is at most the column size.

With this intuition, we can make the DP state smaller. Instead of maintaining the left and right stone boundary position, we can just maintain the distance between the two stones instead. The three possibilities of moving the stone boundaries in the next row (expand, shrink, unchanged) now translate to five possibilities of adding the stone distance, by -2, -1, 0, 1, or 2 as shown in the following examples.

```
prev row:  -*XXX*-  -*XXX*-  -*XXX*-  -*XXX*-  -*XXX*-
next row:  --*X*--  --*XX*-  -*XXX*-  -*XXXX*  *XXXXX*
             (-2)      (-1)       (0)       (1)       (2)
```

Note that for (-1) and (1) there is another possibility for the next row, but both have the same distance between the left and right stone boundaries.

By switching the left and right boundary position to the distance between the left and right stone, we can reduce the DP states to three states (the remaining rows left, stone distance of the previous row, the remaining intersection to enclose). This reduces the complexity to O(**N** * *sqrt*(**N** * **M**) * **N** * **M**). The amortized cost is less than 32 million operations per test case which is fast enough to answer 100 test cases. Below is a sample implementation in Python 3:

```
from functools import lru_cache
import sys

@lru_cache(maxsize = None)  # Memoization.
def rec(rem_rows, prev_dist, rem_points, M):
  if rem_points <= 0:  # If the remaining area is non positive,
    return 0  # then no stone is needed.

  ret = 1000000  # Infinity.
  if rem_rows <= 0:  # No more row but rem_points is still > 0.
    return ret  # Return infinity.

  if M == 1:  # Special case where each row only has one stone.
    return rem_points  #  rem_rows >= rem_points is guaranteed.

  min_dist = max(prev_dist - 2, 1)
  max_dist = min(prev_dist + 2, M)
  for next_dist in range(min_dist, max_dist + 1):
    if next_dist >= rem_points:
      # Close the enclosure for the last row.
      ret = min(ret, next_dist)
    elif next_dist > 1:
      # Cover this row using 2 stones.
      next_rem_points = rem_points - next_dist
      ret = min(ret, \
        2 + rec(rem_rows - 1, next_dist, next_rem_points, M))

  return ret


def min_stones(N, M, K):
  if N < M:  # If the row size is smaller than the column size
```

```
    (N, M) = (M, N)  # Transpose the grid

  res = 1000000
  # Try all possible number of stones for the top row.
  for stones in range(1, min(K, M) + 1):
    # The stones needed to cover the top row + the next rows.
    stones = stones + rec(N - 1, stones, K - stones, M)
    res = min(res, stones)

  return res

sys.setrecursionlimit(5000)
for tc in range(int(input())):
  print("Case #%d: %d" % (tc+1, \
    min_stones(*map(int, input().split()))))
```

Greedy

Intuitively, the two stones in each intermediate row can be greedily placed as far as possible from each other to maximize the area enclosed without adding any additional stone. Consider the following example:

```
---------      ---------      ----*----
---------      ---------      ---*X*---
--******--      --******--      --*XXX*--
--*XXX*--      -*XXXXX*-      -*XXXXX*-
--*XXX*--  ->  *XXXXXXX*  ->  *XXXXXXX*
--*XXX*--      -*XXXXX*-      -*XXXXX*-
--******--      --******--      --*XXX*--
---------      ---------      ---*X*---
---------      ---------      ----*----
   (a)            (b)            (c)
```

The enclosure in Figure (a) can be improved by moving the two stones (in each of the three intermediate rows) as far as possible from each other as shown in Figure (b). Similar reasoning can be made for each column that contain only two stones: move the top and bottom stones as far from each other as possible in that column. The enclosure in Figure (b) can be improved in the same way, resulting the enclosure as shown in Figure (c).

Knowing that the optimal shape resembles a diamond, the greedy approach is to try to construct a diamond-shaped enclosure with area at least **K**. However, this is not always feasible if the grid is not large enough. In such case, the diamond-shape may be â€œtruncatedâ€ at the top / left / right / bottom sides as shown below for the best enclosure for **N** = 6, **M** = 7, **K** = 27

```
--***--
-*XXX*-
*XXXXX*
-*XXXX*
--*XX*-
---**--
```

Another useful observation is that the empty intersections at the corners always forms a right triangle. This allows us to generate all possible truncated (and perfect) diamond by placing empty triangles at the corners. Notice that the sizes (the length of its side) of the empty triangles at the corners may be different by at most one size.

Fortunately, the large input is small enough that we can brute-force for all possible truncated (and perfect) diamond shapes. First we try all possible grid size, and for each possible grid size, we try to put empty triangles at the corners and compute the enclosure size and the stones needed. We record and return the minimum stones needed to construct the shape with area at least **K**. Below is a sample implementation in Python 3:

```
def empty_triangle(size):
  return size * (size + 1) / 2

def min_stones(N, M, K):
  if N > M:
    (N, M) = (M, N)

  best = K
  for R in range(2, N + 1):
    for C in range(R, M + 1):
      if R * C >= K:
        for i in range(2 * R):
          cover = R * C
          cover -= empty_triangle(i // 4)
          cover -= empty_triangle((i + 1) // 4)
          cover -= empty_triangle((i + 2) // 4)
          cover -= empty_triangle((i + 3) // 4)
          if cover < K:
            break
          stones = 2 * (R + C) - 4 - i
          best = min(best, stones)

  return best
```

The complexity of the above solution is O($\mathbf{N}^2$ * **M**). However, if you are well versed in Mathematics, you can further improve the search to O(*log*(**K**)) by doing a binary search on the number of stones needed to form the truncated diamond shape and compute the number of enclosed intersection points in O(1) to make the binary search decision.