- State (in big-O notation) the implementation's memory complexity and briefly explain how you calculated this bound.

  For the constructor, the memory complexity is $O(n^2)$ since if the input of width is 'n' and the input of height is 'n', the overall memory complexity is 'n*n'. So, the memory complexity is $O(n^2)$. For resize method, since when calling this method, a temporary 2D array with the width of 'n' and the height of 'n' will be initialized, the memory complexity of this method is $O(n^2)$. Although after calling this method, the space of the temporary 2D array will be freed, to consider the worst case, the memory complexity is still $O(n^2)$. For add ,get, remove method, the memory complexity is O(1) since if the input is invalid, the 'throw' statement will only execute once, which is a constant. If the input is valid, no objects will be initialized. So, the memory complexity is O(1). For clear method, no objects will be initialized at any time. So, the memory complexity for this method is also O(1). **In conclusion**, the implementation's memory complexity is $O(n^2)$.

- Use this bound to evaluate the overall memory efficiency of your implementation - you should especially consider the case where your grid is very large but has very few elements.

  The overall memory efficiency is $O(n^2)$. For the constructor, when the grid is very large, it means the width and height of the grid tend to be 'n'. In resize method, the worst case is to resize this 2D array's width and height to 'n'. Due to the memory efficiency of other methods is O(1), the worst case of the memory efficiency for this implementation is $2n^2 + a$(which means constants), which is $O(n^2)$.

- Research and describe at least one alternative implementation that has significant advantages over the one chosen.

  Prakash Sharma[1] provides one alternative implementation which is to use a 1D array. A 2D array "A[x][y]" can be easily converted to a 1D array "A[x*y]" in row major order or column major order. To get the desired element in this 1D array, just need to follow this equation "A[x+(width*y)]" to implement it in column major order. For example, if there is a 5*5 2D array "E", and I would like to access the element at E[2][3], after converting it to a 1D array "R[5*5]", I can also get the same element by accessing R[17]

(0,0)	0	(0,1)	5	(0,2)	10	(0,3)	15	(0,4)	20
(1,0)	1	(1,1)	6	(1,2)	11	(1,3)	16	(1,4)	21
(2,0)	2	(2,1)	7	(2,2)	12	(2,3)	17	(2,4)	22
(3,0)	3	(3,1)	8	(3,2)	13	(3,3)	18	(3,4)	23
(4,0)	4	(4,1)	9	(4,2)	14	(4,3)	19	(4,4)	24

  Similarly, I can also implement add, remove, clear and resize method by using 1D array instead.

  Aaron Digulla[2] thinks there are four main advantages to use a 1D array instead of a 2D array. 1.There will be less code, which means it is less time-consuming. 2. Less code means the code looks cleaner and it will help reduce bugs since bugs cannot hide

in few lines of code. 3.Due to less code, it is much easier to replace your code if you do something wrong. 4.Simple code can lead to more simple code.

- Compare the alternative implementation with your chosen approach in terms of both memory and runtime efficiency

  Constructor:

  Memory

  1D- Since it only has “n” elements, the memory efficiency is O(n).

  2D- Since it has “n*n” elements, the memory efficiency is $O(n^2)$.

  Runtime

  1D- Since it only runs once and is irrelevant with the input size, the runtime efficiency is O(1).

  2D- Same as 1D, also O(1).

  Add:

  Memory

  1D-O(1)

  2D-O(1)

  Runtime

  1D-O(1)

  2D-O(1)

  Remove:

  Memory

  1D-O(1)

  2D-O(1)

  Runtime

  1D-O(1)

  2D-O(1)

  Clear:

Memory

1D-O(1)

2D-O(1)

Runtime

1D-Since only iterating “n” elements, the runtime efficiency is O(n)

2D-Since iterating “n*n” elements, the runtime efficiency is $O(n^2)$

Resize:

Memory

1D-Since initialising a temporary array with the size of “n”, the memory efficiency is O(n).

2D-Since initialising a temporary array with the size of “n*n”, the memory efficiency is $O(n^2)$.

Runtime

1D-Since iterating “n” elements, the time efficiency is O(n).

2D-Since iterating “n*n” elements, the time efficiency is $O(n^2)$.

Overall: for the worst case

Memory

1D-O(n)

2D-$O(n^2)$

Runtime

1D-O(n)

2D-$O(n^2)$

Reference:

1. Prakash Sharma. (Feb. 24,2017). What does 'creating a two-dimensional array using a one-dimensional array' mean? Retrieved from: https://www.quora.com/What-does-creating-a-two-dimensional-array-using-a-one-dimensional-array-mean

2. Aaron Digulla. (Apr. 09, 2009). Implementing a matrix, which is more efficient - using an Array of Arrays (2D) or a 1D array? Retrieved from: https://stackoverflow.com/questions/732684/implementing-a-matrix-which-is-more-efficient-using-an-array-of-arrays-2d-o/732688#732688