Analysis of the MirrorMazeBruteForce Algorithm

# Time complexity

## Best cases:

First, we analyze the time complexity of each of the methods that getSolution() invokes, after that we add those time complexities in order to get the time complexity of the getSolution() method.

getStartingPosition():

Ω(1)

getMirrorPositions(maze) - Ω(width\*height), if width = m and height=n, then we can write:

Ω (m\*n)

getCombinations(mirrors.length) - Ω (2mirrorNumber\*mirrorNumber), where mirrorNumber=0 in the best case:

Ω ( 1)

getSolution()(lines 52 up to 64)- Ω (2mirrorNumber \*(mirrorNumber)), where mirrorNumber = 0 in the best case, as discussed previously, so:

Ω ( 1)

Finally, we add these time complexities to get the time complexity of getSolution() method.

C + m\*n

C – constant which represents the constant time complexity Ω(1) of the getStartingPosition() method

## Worst cases:

The same logic is used when describing worst case time complexity. First, we describe time complexity of each method that getSolution() invokes, then we add time complexities of these methods in order to get the time complexity of the getSolution() method.

getStartingPosition():

Ω(2\*(m+n)-8)

if height=width, and height=n, width=m, and we represent them as “n”, then we can generalize time complexity:

Ω(4n-8)

getMirrorPositions(maze):

Ω(m\*n)

if width=height, and we represent them as “n”, then we can generalize time complexity:

Ω(n2)

getCombinations(mirrors.length)- O(2mirrorNumber\*mirrorNumber), where mirrorNumber=width\*height-2\*(width+height) in the worst scenario, so we can write it:

Ω ( 2m\*n-2\*(m+n) \* (m\*n-2\*(m+n)))

getSolution()(lines 52 up to 64)- O(2mirrorNumber \*(mirrorNumber+(height-2)\*(width-2)+2)), where mirrorNumber in the worst scenario is the same as above

Ω ( 2m\*n-2\*(m+n)\*(m\*n-2\*(m+n)+(n-2)\*(m-2)+2))

Finally, we add these time complexities to get the time complexity of getSolution() method:

Ω(m\*n)+ Ω(n2)+ Ω ( 2m\*n-2\*(m+n) \* (m\*n-2\*(m+n)))+ Ω ( 2m\*n-2\*(m+n)\*(m\*n-2\*(m+n)+(n-2)\*(m-2)+2))

# Space complexity

As in the case of the analysis of time complexity, we first analyze the space complexity of each of the methods that getSolution() invokes, after that we add those space complexities in order to get the space complexity of the getSolution() method. Interestingly, the mathematical function that expresses the space complexity of this algorithm is always the same(the same function applies to both the worst and the best case).

getStartingPosition():

Ω(1)

O(1)

getMirrorPositions(maze) - Ω(mirrorNumber), the function of the worst case is the same as well, where mirrorNumber=width\*height-2\*(width+height) in the worst scenario, if width = m and height = n, then we can write the space complexity function for the best case and the worst of this method as:

Ω ((m\*n-2\*(m+n)))

O((m\*n-2\*(m+n)))

getCombinations(mirrors.length) - Ω (2mirrorNumber), the function of the worst case is the same as well, where mirrorNumber=width\*height-2\*(width+height) in the worst scenario, if width = m and height = n, then we can write the space complexity function for the best case and the wors case of this method as:

Ω ( 2m\*n-2\*(m+n))

O(2m\*n-2\*(m+n))

getSolution()(lines 52 up to 64):

Ω (1)

O(1)

Finally, we add these space complexities to get the space complexity of getSolution() method for the worst case and the base case.

C + m\*n-2\*(m+n) + 2m\*n-2\*(m+n) + C1

C – constant number which represents the constant space complexity of the getStartingPosition() method

C1 – constant number which represents the constant space complexity of the getSolution()(lines 52 up to 64)