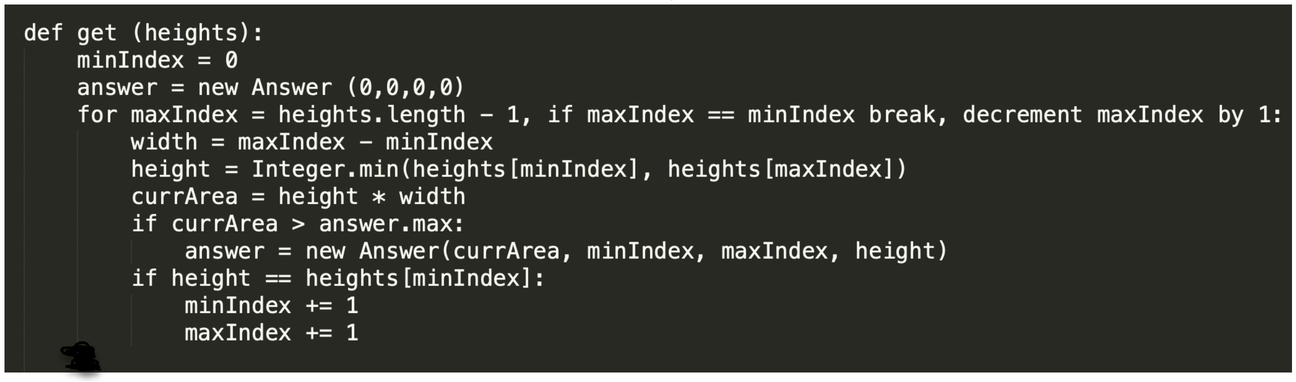
Max Rectangle
Tom Bohbot
April 2021

1) Brute force approach and Big O performance:
   The brute force approach would be to calculate the area of every element in the array to every other element in the array. This approach would be O(n^2) as it would involve an outer for loop to iterate through each element in the array and an inner for loop that iterates through every other element in the array and finds the area of the outer for loops element and inner for loop. Then there would be a maxArea variable instantiated to zero before the first loop and whenever a new area exceeds the maxArea variable it would become the maxArea. By the time the outer for loops finishes every possible combination of points would be exhausted and the all area would be looked at, ensuring that maxArea is indeed the maxArea.

2) Describe the core idea of your algorithm:
   Disregarding heights, the biggest area of a rectangle should intuitively have the width of both extremities of the x points. Therefore I set a left and right pointer and get the current area calculated through finding the min height of those two points and multiplying it by the width. If that current area is greater than my max area then I set it to be the max area. Afterwards I shift one unit over of the point that had the min. height as its area has already been calculated. Once either pointer has seen all points in the array then that means that all potential max areas have been seen and I can return my currently calculated max area as the answer.
   How my algorithm is greedy:
   My algorithm never back tracks nor realized that it made a mistake. The core idea of my algorithm is to replace the max area found every time an area is found to be larger in one loop, it will never decide it made a mistake and go to a previously found area.

3) Pseudo Code:

Pseudocode assumes that an answer class is already made.



```
def get (heights):
    minIndex = 0
    answer = new Answer (0,0,0,0)
    for maxIndex = heights.length - 1, if maxIndex == minIndex break, decrement maxIndex by 1:
        width = maxIndex - minIndex
        height = Integer.min(heights[minIndex], heights[maxIndex])
        currArea = height * width
        if currArea > answer.max:
            answer = new Answer(currArea, minIndex, maxIndex, height)
        if height == heights[minIndex]:
            minIndex += 1
            maxIndex += 1
```

4) Proof of feasibility and optimality:
Direct Proof:
The area of a rectangle can be defined through the height multiplied by the width. The larger the width and height, the larger the area will be. The algorithm starts by comparing points at opposing ends and then looks through more central points if a larger area can be found. Since the algorithm starts by analyzing both ends, the maximum width is initially analyzed. However, since heights play a role as well, the height of a certain point can compensate for a smaller width. Every time my algorithm searches for a new area, one of the pointers will stagnate while the other will approach one unit closer to the stagnated pointer. The pointer that will shift is the one with the smaller height. Since the width will be reduced by one unit (as the smaller pointer approaches by one unit), there is no reason to take the higher pointer as it has some unused height potential since the algorithm is constricted by the smaller of the two heights. Every time one of the pointers approaches the other the area will be recalculated. If the new set of pointers have larger y values then they may potentially have a larger area as the width only decreased by one but the height does not have a fixed change. If an area is found to be larger than the previous largest area then it becomes the newest maximum area. The maximum area is returned once both pointers are pointing to the same index as that means all options have been exhausted. All the options must have been exhausted because the width will now become zero causing an area to be zero, and if the pointers overlap then there will be a repetition of previous answers. This algorithm visits every element only once which must be the quickest solution as it is impossible to be certain of the max area without visiting all elements since the height (or y value) of an element cannot be predicted. Due to the fact that this algorithm visits every element only once it causes the algorithm to run in O(n) time.
The proof of feasibility can be proved through construction since the pseudocode shows that a rectangle will always be formed. Due to the fact that every potential larger area is calculated until both index pointers are equal, the algorithm is sure to be optimal as no larger area could exist. End of proof.