**BEGINNING**

**You are working in a 2d coordinate system.**
**You are given the following classes.**

C++:
```
Class Point{
double x;
double y;
Point(x,y);
}

Class Rectangle{
// a, b refer to opposing points of the rectangle
Point a;
Point b;
Rectangle(Point a, Point b);
}
```

Python:
```
class Point():
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Rectangle():
    def __init__(self, a, b):
        self.a = a
        self.b = b
```

**Part 1:**
**a. Write a function that takes in a rectangle and a point, and returns rather or not the point lies within the rectangle**

Approach: find the range of x and y of the rectangle, and then determine if the point's x and y are within the range

```
def pointWithinRectangle(pointC, rectangle):
    pointA = rectangle.a
    pointB = rectangle.b

    #Edge case: rectangle has zero of 0. Meaning, there is no rectangle.
    if abs(pointA.x - pointB.x) * abs(pointA.y - pointB.y) == 0:
        return False

    xMin = min(pointA.x, pointB.x)
    xMax = max(pointA.x, pointB.x)
```

```
    yMin = min(pointA.y, pointB.y)
    yMax = max(pointA.y, pointB.y)

    dimensionsWithin = 0

    if xMin <= pointC.x <= xMax:
        dimensionsWithin += 1
    if yMin <= pointC.y <= yMax:
        dimensionsWithin += 1

    return dimensionsWithin == 2
```

**b. Write a function that takes in two rectangles and returns rather or not the rectangles overlap.**

Approach: find the rectangle with the bigger area, and then look for the smaller rectangle within the bigger rectangle.It would not be logical to look for the bigger rectangle within the smaller one

I do consider having only one point in common to be an overlap, except for the area == 0 case (no rectangle)

```
def overlapRectangles(rectangle1, rectangle2):
        point1A = rectangle1.a
        point1B = rectangle1.b

        point2A = rectangle2.a
        point2B = rectangle2.b

        area1 = abs(point1A.x - point1B.x) * abs(point1A.y - point1B.y)
        area2 = abs(point2A.x - point2B.x) * abs(point2A.y - point2B.y)

        #If one or both areas are == 0, no overlap possible.
        if area1 == 0 or area2 == 0:
            return False

        if area1 < area2:
            return pointWithinRectangle(point1A, rectangle2) or pointWithinRectangle(point1B,
    rectangle2)
        else:
            return pointWithinRectangle(point2A, rectangle1) or
    pointWithinRectangle(point2B, rectangle1)
```

**PART 2**

The classes are modified to store integer coordinates:

```
Class Point{
int x;
int y;
Point(x,y);
}

Class Rectangle{
// a, b refer to opposing points of the rectangle
Point a;
Point b;
Rectangle(Point a, Point b);
}
```

**You are given an 2d array of doubles representing a set of points.**

**a.**
**Write a function that returns the sum of all doubles within a rectangle in the 2d array.**

```
Double rectangleSum(double[][] values, Rectangle rect){
}
```

Interpretation: Size of values[i] == 2 for all i; values[i][0] is the x and values[i][1] is the y. The size of values is the number of points. We turn the doubles into an int.

```
def rectangleSum(values, rect):
        doubleSum = 0
        for value in values:
                #int removes the decimal part, like what happens if we put a double into a c++ int
                #variable
                point = Point( int(value[0]), int(value[1]) )
                if pointWithinRectangle(point, rect):
                        doubleSum += value[0] + value[1]
        return doubleSum
```

**b. What is the running time complexity of your algorithm?**

pointWithinRectangle is constant. I cannot find documentation on int()'s runtime, but I assume it would vary depending on the input. I do not include int()'s potential runtime below.:

Linear runtime with the size of values

**What is the memory complexity of your algorithm?**
Constant space. Python has automatic garbage collection.

**After the function you wrote has been deployed, you find that it is often used multiple times on the same array but with different rectangles. Your coworker suggests that you can make the solution much more efficient if you have the option of preprocessing the 2d array.**

**c. Come up with a strategy for preprocessing the array and implementing it?**
1. I can create another function to preprocess the values array and turn all its elements to point, and then we would simply need to loop through the array to check if the points are in the rectangle. Then we could just use that array for further needs.

2. I can use a higher order function to preprocess the values array and return a function that works only with that preprocessed array.

Both ideas are pretty similar. I'll go with 2.

If we're really trying to make it as fast as possible, the only thing that can be preprocessed is the point = Point( int(point[0]), int(point[1])). I would create another array (just 1D), that stores every point from the 2d array. In subsequent calls with different rectangles, I would just use the 1D array instead. This would require extra space linear to the size of the 2d array but should save some time (in terms of actual runtime not Big O) with not needing to call int and create the same Points again with the same 2d array.

```
def processArray(values):
        points = []
        for value in values:
                point = Point( int(value[0]), int(value[1]) )
                points.append(point)

        def rectangleSum(rect):
                doubleSum = 0
                for i in range(len(points)): #size of points = size of values
                        if pointWithinRectangle(points[i], rect):
                                #the doubles are in values not points
                                doubleSum += values[i][0] + values[i][1]
                return doubleSum

        return rectangleSum
```

Not sure how higher order functions in C++ work but just in case it's different, it would work like this:

Call processArray with a 2D array. This would create the points array and process the 2D array and return the function rectangleSum, say we save this function into variable rSum. Then we can use this function with any rect to do the summing. So for every future call to the rSum function, we have already preprocessed the 2D array.

**d. What is the running time complexity of your algorithm?**

Linear runtime with the size of values minus the time it takes to call int() twice for every array within values

**What is the memory complexity of your algorithm?**
Linear with size of values, since we create another array that contains the preprocessed points

**e. How do the changes impact the performance of rectangleSum?**
This implementation doesn't have much to preprocess, so still Linear runtime with the size of values minus the time it takes to call int() twice and time to call Point() for every array within values. This should save some amount of time (in terms of actual run time not big O runtime, though again, I cannot find documentation on the runtime of int(), but I assume it's not constant for BIg O) especially with multiple calls using the same 2d array.

---

**INITIAL INTERPRETATION FOR PART 2**

**a.**
**Write a function that returns the sum of all doubles within a rectangle in the 2d array.**

Below was my initial thought. After some more thinking, I feel like there would be no point in values being 2d (it could just be 1d), and what about cases like 1.10? The 0 wouldn't be stored. So this probably wasn't right.

I see that the classes have been modified to store ints. I'm assuming each double #should represent a point but points should have two numbers. Thus, I can split the #double into two parts, the number before the decimal, and the number after. This would #be two ints, which can be a point.

```
def rectangleSum(values, rect):
        doubleSum = 0
        for row in values:
                for double in row:
                        #1.2234 would become 1 and 2234
                        doubleSplit = str(double).split('.')
                        point = Point(doubleSplit[0], doubleSplit[1])
                        if pointWithinRectangle(point, rect):
                                doubleSum += double
        return doubleSum
```

**b. What is the running time complexity of your algorithm?**

M = num rows
N = num cols
K = str(double).split('.') which is linear runtime to the string length

runtime = M*N*K

**What is the memory complexity of your algorithm?**
Constant space. Python has automatic garbage collection.

**c. Come up with a strategy for preprocessing the array and implementing it?**

```
import math
def processArray(values):
    for r in range(len(values)): #need to use indices here otherwise wouldn't modify
        for c in range(len(values[0]))::
            #1.2234 would become 1 and 2234
            doubleSplit = str(double).split('.')
            values[r][c] = Point(doubleSplit[0], doubleSplit[1])

    def rectangleSum(rect):
        doubleSum = 0
        for row in values:
            for point in row:
                if pointWithinRectangle(point, rect):
                    numDigits = int(math.log10(point.y))+1
                    doubleSum += point.x + (point.y / numDigits)
                    #the above turns a point back into decimal
        return doubleSum
    return rectangleSum
```

**d. What is the running time complexity of your algorithm?**
The running time of rectangle sum in this case would just be M*N, M=num rows, N=num cols with values already being processed.

**What is the memory complexity of your algorithm?**
Constant. (Unless for some reason we don't want to modify the values array, then need M*N to hold all a copy of values except with points)

**e. How do the changes impact the performance of rectangleSum?**
It goes from M*N*K every time to K time for the first call to processArray, and then M*N time for every subsequent call to rectangleSum. Given the same values array, processArray is only needed to be called once. It returns a function rectangleSum that takes in a rect that can be used for all rectangles to calculate the sum in M*N time with the array already preprocessed.

---

**UNIT TESTS**

```
point1 = Point(-4, -1)
point2 = Point(-2, -4)
point3 = Point(-5,1)
point4 = Point(-3, -2)
point5 = Point(-4.1, -1)
point6 = Point(-6.5,4)
point7 = Point(7, 1)
point8 = Point(10, 11)

rectangle1 = Rectangle(point1, point2)
rectangle2 = Rectangle(point1, point3)
rectangle3 = Rectangle(point1, point4)
rectangle4 = Rectangle(point5, point6)
rectangle5 = Rectangle(point7, point8)
```

**pointWithinRectangle tests**

If my aim was to make more readable tests, I would write the following below.

```
'True, (-2,-1) is within the rectangle formed by (-4, -1) and (-2,-4): ' +
str(pointWithinRectangle(Point(-2, -1), rectangle1)))
```

Otherwise I would just aim to use my time to write more tests instead of formatting each one.

```
print('pointWithinRectangle Tests')
print(pointWithinRectangle(Point(-2,-1), rectangle1)) #edge
print(pointWithinRectangle(Point(-3,-3), rectangle1)) #center
print(pointWithinRectangle(Point(-4,-4), rectangle1)) #edge
print(pointWithinRectangle(Point(1,5), rectangle1) == False)   #outside
print(pointWithinRectangle(Point(-1,-1), rectangle1) == False) #barely outside
```

Afer this round of tests, I am fairly certain my code works in most if not all cases even in different quadrants and if decimals as I understand how my code works.

**overlapRectangles Tests**

```
print('overlapRectangles Tests')
print(overlapRectangles(rectangle1, rectangle2)) #shares a point and ONLY that point is overlapping
print(overlapRectangles(rectangle1, rectangle3)) #shares more than a point in common
print(overlapRectangles(rectangle1, rectangle4) == False) #barely not overlapping
print(overlapRectangles(rectangle1, rectangle5) == False) #nowhere close to overlapping
```

I did not create tests for Part 2 as I've checked multiple times, and there's no logical error possible. If this were a real project, I would be sure to check everything.