20.12 Given an NxN matrix of positive and negative integers, write code to find the sub-matrix with the largest possible sum.

pg 92

SOLUTION

Brute Force: Complexity O(N^6)

Like many "maximizing" problems, this problem has a straight forward brute force solution. The brute force solution simply iterates through all possible sub-matrixes, computes the sum, and finds the biggest.

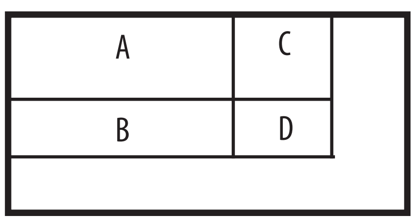
To iterate through all possible sub-matrixes (with no duplicates), we simply need to iterate through all order pairings of rows, and then all ordered pairings of columns.

This solution is O(N^6), since we iterate through O(N^4) sub-matrixes, and it takes O(N^2) time to compute the area of each.

Optimized Solution: O(N^4)

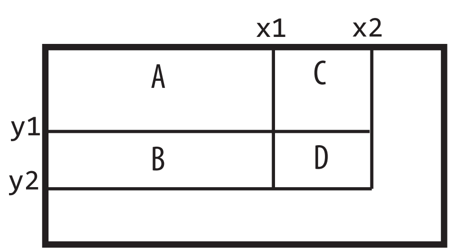
Notice that the earlier solution is made slower by a factor of O(N^2) simply because computing the sum of a matrix is so slow. Can we reduce the time to compute the area? Yes! In fact, we can reduce the time of computeSum to O(1).

Consider the following:



If we had the sum of the smaller rectangle (the one including A, B, C, D), and we could compute the sum of D as follows: area(D) = area(A through D) - area(A) - area(B) - area(C).

What if, instead, we had the following:



with the following values (notice that each Val_* starts at the origin):

```
Val_D = area(point(0, 0) -> point(x2, y2))
Val_C = area(point(0, 0) -> point(x2, y1))
Val_B = area(point(0, 0) -> point(x1, y2))
Val_A = area(point(0, 0) -> point(x1, y1))
```

With these values, we know the following:

```
area(D) = Val_D - area(A union C) - area(A union B) + area(A).
```

Or, written another way:

```
area(D) = Val_D - Val_B - Val_C + Val_A
```

Can we efficiently compute these Val_* values for all points in the matrix? Yes, by using similar logic:

```
Val_(x, y) = Val(x - 1, y) + Val(y - 1, x) - Val(x - 1, y - 1)
```

We can precompute all such values, and then efficiently find the maximum submatrix. See the following code for this implementation

```
public static int getMaxMatrix(int[][] original) {
    int maxArea = Integer.MIN_VALUE; // Important! Max could be < 0
    int rowCount = original.length;
    int columnCount = original[0].length;
    int[][] matrix = precomputeMatrix(original);
    for (int row1 = 0; row1 < rowCount; row1++) {
        for (int row2 = row1; row2 < rowCount; row2++) {
            for (int col1 = 0; col1 < columnCount; col1++) {
                for (int col2 = col1; col2 < columnCount; col2++) {
                    maxArea = Math.max(maxArea, computeSum(matrix,
                        row1, row2, col1, col2));
                }
            }
        }
    }
    return maxArea;
}

private static int[][] precomputeMatrix(int[][] matrix) {
    int[][] sumMatrix = new int[matrix.length][matrix[0].length];
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix.length; j++) {
            if (i == 0 && j == 0) { // first cell
                sumMatrix[i][j] = matrix[i][j];
            } else if (j == 0) { // cell in first column
                sumMatrix[i][j] = sumMatrix[i - 1][j] + matrix[i][j];
            } else if (i == 0) { // cell in first row
                sumMatrix[i][j] = sumMatrix[i][j - 1] + matrix[i][j];
            } else {
                sumMatrix[i][j] = sumMatrix[i - 1][j] +
                  sumMatrix[i][j - 1] - sumMatrix[i - 1][j - 1] +
```

```
                    matrix[i][j];
            }
        }
    }
    return sumMatrix;
}

private static int computeSum(int[][] sumMatrix, int i1, int i2,
                              int j1, int j2) {
    if (i1 == 0 && j1 == 0) { // starts at row 0, column 0
        return sumMatrix[i2][j2];
    } else if (i1 == 0) { // start at row 0
        return sumMatrix[i2][j2] - sumMatrix[i2][j1 - 1];
    } else if (j1 == 0) { // start at column 0
        return sumMatrix[i2][j2] - sumMatrix[i1 - 1][j2];
    } else {
        return sumMatrix[i2][j2] - sumMatrix[i2][j1 - 1]
               - sumMatrix[i1 - 1][j2] + sumMatrix[i1 - 1][j1 - 1];
    }
}
```