

Max Sum of Rectangle No Larger Than K

Given a non-empty 2D matrix *matrix* and an integer *k*, find the max sum of a rectangle in the *matrix* such that its sum is no larger than *k*.

Example:

```
Given matrix = [  
  [1, 0, 1],  
  [0, -2, 3]  
]  
k = 2
```

The answer is 2. Because the sum of rectangle `[[0, 1], [-2, 3]]` is 2 and 2 is the max number no larger than k (k = 2).

Note:

1. The rectangle inside the matrix must have an area > 0.
2. What if the number of rows is much larger than the number of columns?

Credits:

Special thanks to [@fujiaozhu](#) for adding this problem and creating all test cases.

Solution 1

The naive solution is brute-force, which is $O((mn)^2)$. In order to be more efficient, I tried something similar to Kadane's algorithm. The only difference is that here we have upper bound restriction K . Here's the easily understanding video link for the problem "find the max sum rectangle in 2D array": [Maximum Sum Rectangular Submatrix in Matrix dynamic programming/2D kadane](#) (Trust me, it's really easy and straightforward).

Once you are clear how to solve the above problem, the next step is to find the max sum no more than K in an array. This can be done within $O(n \log n)$, and you can refer to this article: [max subarray sum no more than k](#).

For the solution below, I assume that the number of rows is larger than the number of columns. Thus in general time complexity is $O[\min(m,n)^2 * \max(m,n) * \log(\max(m,n))]$, space $O(\max(m, n))$.

```
int maxSumSubmatrix(vector<vector<int>>& matrix, int k) {
    if (matrix.empty()) return 0;
    int row = matrix.size(), col = matrix[0].size(), res = INT_MIN;
    for (int l = 0; l < col; ++l) {
        vector<int> sums(row, 0);
        for (int r = l; r < col; ++r) {
            for (int i = 0; i < row; ++i) {
                sums[i] += matrix[i][r];
            }

            // Find the max subarray no more than K
            set<int> accuSet;
            accuSet.insert(0);
            int curSum = 0, curMax = INT_MIN;
            for (int sum : sums) {
                curSum += sum;
                set<int>::iterator it = accuSet.lower_bound(curSum - k);
                if (it != accuSet.end()) curMax = std::max(curMax, curSum - *it);
                accuSet.insert(curSum);
            }
            res = std::max(res, curMax);
        }
    }
    return res;
}
```

written by [java-chao](#) original link [here](#)

Solution 2

```
/* first consider the situation matrix is 1D
   we can save every sum of 0~i(0<=i<len) and binary search previous sum to find
   possible result for every index, time complexity is O(NlogN).
   so in 2D matrix, we can sum up all values from row i to row j and create a 1D
   array
   to use 1D array solution.
   If col number is less than row number, we can sum up all values from col i to
   col j
   then use 1D array solution.
*/
public int maxSumSubmatrix(int[][] matrix, int target) {
    int row = matrix.length;
    if(row==0)return 0;
    int col = matrix[0].length;
    int m = Math.min(row,col);
    int n = Math.max(row,col);
    //indicating sum up in every row or every column
    boolean colIsBig = col>row;
    int res = Integer.MIN_VALUE;
    for(int i = 0;i<m;i++){
        int[] array = new int[n];
        // sum from row j to row i
        for(int j = i;j>=0;j--){
            int val = 0;
            TreeSet<Integer> set = new TreeSet<Integer>();
            set.add(0);
            //traverse every column/row and sum up
            for(int k = 0;k<n;k++){
                array[k]=array[k]+(colIsBig?matrix[j][k]:matrix[k][j]);
                val = val + array[k];
                //use TreeMap to binary search previous sum to get possible resu
                Integer subres = set.ceiling(val-target);
                if(null!=subres){
                    res=Math.max(res,val-subres);
                }
                set.add(val);
            }
        }
    }
    return res;
}
```

written by [wuzixigua](#) original link [here](#)

Solution 3

I got a TLE for the Python code below, because the time cost of `bisect.insort` is $O(n)$ for a built-in list.

The code was rejudged as accepted just now, but very slow... 1800ms+

```
class Solution(object):
    def maxSumSubmatrix(self, matrix, k):
        """
        :type matrix: List[List[int]]
        :type k: int
        :rtype: int
        """
        m = len(matrix)
        n = len(matrix[0]) if m else 0

        M = max(m, n)
        N = min(m, n)
        ans = None
        for x in range(N):
            sums = [0] * M
            for y in range(x, N):
                slist, num = [], 0
                for z in range(M):
                    sums[z] += matrix[z][y] if m > n else matrix[y][z]
                    num += sums[z]
                    if num <= k: ans = max(ans, num)
                    i = bisect.bisect_left(slist, num - k)
                    if i != len(slist): ans = max(ans, num - slist[i])
                    bisect.insort(slist, num)
        return ans or 0
```

Could anybody share a more efficient Python solution? Thank you :D

written by [%E5%9C%A8%E7%BA%BF%E7%96%AF%E7%8B%82](#) original link [here](#)

From [LeetCoder](#).