1 UVA Problem 108: Maximum Sum

Background

A problem that is simple to solve in one dimension is often much more difficult to solve in more than one dimension. Consider satisfying a boolean expression in conjunctive normal form in which each conjunct consists of exactly 3 disjuncts. This problem (3-SAT) is NP-complete. The problem 2-SAT is solved quite efficiently. In contrast, some problems belong to the same complexity class regardless of the dimensionality of the problem.

The Problem

Given a 2-dimensional array of positive and negative integers, and the sub-rectangle with the largest sum. The sum of a rectangle is the sum of all the elements in that rectangle. In this problem the sub-rectangle with the largest sum is referred to as the maximal sub-rectangle . A sub-rectangle is any contiguous sub-array of size 1x1 or greater located within the whole array.

Input and Output

The input consists of an NxN array of integers. The input begins with a single positive integer N on a line by itself indicating the size of the square two dimensional array. This is followed by N^2 integers separated by white-space (newlines and spaces). These N^2 integers make up the array in row-major order (i.e., all numbers on the first row, left-to-right, then all numbers on the second row, left-to-right, etc.). N may be as large as 100. The numbers in the array will be in the range [-127, 127]. The output is the sum of the maximal sub-rectangle.

1.1 Mathematical Formulation

Given an input of N^2 numbers in the range of [-127, 127] in the form of an NxN matrix, lets call this matrix E. Given E, we will deterermine the maximal sub-rectangle. We define a sub-rectangle as being synonymous to a sub-matrix (E*) of E. i.e

$$E = \begin{bmatrix} e_{1,1} & e_{1,2} & \cdots & e_{1,N} \\ e_{2,1} & e_{2,2} & \cdots & e_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ e_{N,1} & e_{N,2} & \cdots & e_{N,N} \end{bmatrix}, E^* = \begin{bmatrix} e_{i,j} & \cdots & e_{i,n} \\ \vdots & \ddots & \vdots \\ e_{m,j} & \cdots & e_{m,n} \end{bmatrix}$$

Where $1 \leq i \leq m \leq N$ and $1 \leq i \leq m \leq N$, where each one's value is defined as $\sum_{x=i}^{m} \sum_{y=j}^{n} e_{x,y} = S$. We will determine the maximal of these S.

1.2 Solution

Important Confusing Data Structures:

- int[N][N] **map** : Used to quick compute sub-rectangles, see below for details
- int **colSum** : Stores the value the elements being examined currently above and including the target element $e_{i,j}$
- int **sum** : Intermediate value stores the sum of possible preceeding colomns on the same row which have potential to combine with the current row.
- int **max** : Stores the value of the maximal sub-rectangle seen thusfar.

Given E as described above from input, we begin our algorithm by creating a quick sum matrix (map) for the input. We do this by, for each element $e_{i,j}$ we sum it with all of the elements directly above it in the same column. i.e. we let:

$$map = \begin{bmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,N} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ s_{N,1} & s_{N,2} & \cdots & s_{N,N} \end{bmatrix}$$

Where every element: $s_{i,j} = \sum_{k=1}^{i} e_{k,j}$ for $1 \leq i, j \leq N$

Algorithm 1 Build sum

```
procedure Main(Scanner in)
    $N \leftarrow$ in.nextInt()
    $map \leftarrow$ initialize
    for $row \in 1..N$ do
        for $col \in 1..N$ do
            map[row][col] = in.nextInt() + map[row-1][col] // fill in map
    print(maxSum(map, N)) // see below
```

We now will implement the main portion of our algorithm which will in turn calculate the value associated with the maximal sub-rectangle. It does this by looking at every potential height (h) of sub-rectangle, where $1 \leq h \leq N$. And considering every valid bottom right element i.e.

$$E^* = \begin{bmatrix} e_{i,j} & \dots & e_{i,n} \\ \vdots & \ddots & \vdots \\ e_{m,j} & \dots & e_{m,n} \end{bmatrix},\ h = (m-i)+1 \textit{ and the bottom right corner is } e_{m,n}$$

We determine the width of the rectangle through dynamic programming. The rule that we use is simple: if the width of the current sum of E^* is greater than 0, we will continue to extend our rectangle, otherwise this is the new left most column. At the end of each decision we check to see if our current value associated with E^* is greater than our currently stored greatest value, if it is, we update and continue otherwise we simply continue.

Algorithm 2 Compute Maximal Sum

```
procedure maxSum(int[][] map, int N)
    $colSum, max \leftarrow$ Integer.MIN_VALUE
    for $h \in 1..N$ do
        $sum \leftarrow 0$
        for $row \in h..N$ do
            for $col \in 1..N$ do
                $colSum \leftarrow$ map[row][col] - map[row-h][col]
                if $sum \geq 0$ then
                    sum += colSum
                else
                    sum = colSum
                if $sum > max$ then
                    $max \leftarrow$ sum
    return max
```

1.3 Correctness

Lemma 1.

Suppose we are given a sum matrix map from the original matrix E (both described above), a height h, and any position $s_{i,j}$ on the map. Then we define this as:

$$\sigma(i,j,h) = \sum_{k=(i-h)+1}^{i} e_{k,j} \implies valueOf \begin{bmatrix} e_{(i-h)+1,j} \\ e_{(i-h)+2,j} \\ \vdots \\ e_{i,j} \end{bmatrix}$$

(Note: the height of this is matrix is h)

Proposition 2.

Given the sum matrix map of the original matrix E and the σ function from Lemma 1 we can determine the maximal sub-rectangle of E of arbitrary height.

Proof.

To proove this we use induction by prooving that we can find the maximal sub-rectangle of height $h, 1 \leq h \leq N$. We do this through a construction proof. In our algorithm we have 3 variables to store pertinant information in: colSum, sum and max. For our proof we will use the same three having them represent the same things as described at the begining of the Solution section. We will use the following rules then for updating sum and max:

$$sum(i,j,h) = max \begin{cases} sum(i,j-1,h) + colSum(i,j,h) \\ colSum(i,j,h) \end{cases} , \; max(i,j) = max \begin{cases} max(i,j-1) \\ sum(i,j,h) \end{cases}$$

It can be noted here that i, j are the row, column coordinates and h is the height as mentioned before. The value of sum and max however never get stored as we only need to keep track of one instance at a time which is the reason for them able to be represented with one variable each. Also sum(i, 1, h) = colSum(i, j, h) and max(i, 1) = max(i-1, N).

For the first step of induction we will let <u>h = 1</u>, we begin with $max = -128, sum = 0,\ and\ colSum(1,2,1) = \sigma(1,1,1) \implies$ sum(1,1,1) = max(1,1) = $\sigma(1,1,1)$ then we move onto the next element and $colSum(1,2,1) = \sigma(1,2,1)$. Using the relationships:

$$sum(1,2,1) = max \begin{cases} sum(1,1,1) + colSum(1,2,1) \\ colSum(1,2,1) \end{cases} , \; max(1,2) = max \begin{cases} max(1,1) \\ sum(1,2,1) \end{cases}$$

As we can see we would have updated our highest sum seen thusfar for this row and the max seen thus-far. Therefore continuing on we will update these at each stage giving us the proper sum seen thus far for the specified height in a specified row and therefore it trivially follows we can keep track of the largest in total. □

1.4 Analysis

For the following analysis, we will say that..

Proposition 3. *The* <u>*space complexity*</u> *of this algorithm is* $\boldsymbol{O(N^2)}$

Proof.

This is due to the fact that all of our data is stored in one NxN int array and 3 integer variables, which yeilds $N^2 + 3$

Giving us a space complexity of $\mathbf{O(N^2)}$

□

Proposition 4. *The* <u>*time complexity*</u> *of this algorithm is* $\boldsymbol{O(N^3)}$

Proof. This is the case because our algorithm will first fill out our map NxN array (N^2) and then runs through a triple for loop. This is, for each height $h \in 1..N$, we run N, (N - h) times. yeilding $[(N \cdot N) + ((N-1) \cdot N) + \cdots + (1 \cdot N)] = N \cdot [\frac{N \cdot (N+1)}{2}]$

Simplifying to a time complexity of $\mathbf{O(N^3)}$

□

1.5 An Example

Let the input be $N = 2$ and our given table be denoted as E:

$$E = \begin{bmatrix} 1 & -2 \\ 3 & 3 \end{bmatrix} \implies map = \begin{bmatrix} 1 & -2 \\ 4 & 1 \end{bmatrix}$$

Now the algorithm will look at the segments with height 1 i.e.

$$Step\ 1 : \begin{bmatrix} \underline{\mathbf{1}} & -2 \\ 4 & 1 \end{bmatrix},\ Step\ 2 : \begin{bmatrix} 1 & \underline{\mathbf{-2}} \\ 4 & 1 \end{bmatrix},\ Step\ 3 : \begin{bmatrix} 1 & -2 \\ \underline{\mathbf{4}} & 1 \end{bmatrix},\ Step\ 4 : \begin{bmatrix} 1 & -2 \\ 4 & \underline{\mathbf{1}} \end{bmatrix}$$

Next we look at the segments with height 2:

$$Step\ 5 : \begin{bmatrix} \mathbf{1} & -2 \\ \underline{\mathbf{4}} & 1 \end{bmatrix},\ Step\ 6 : \begin{bmatrix} 1 & \mathbf{-2} \\ 4 & \underline{\mathbf{1}} \end{bmatrix}$$

Which we can see below how the values change; note when we go to a new row, we reset the value of sum to 0.

Variable	Height = 1				Height = 2	
	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
colSum	1 $\implies$ 1	-2 $\implies$ -2	4 - 1 $\implies$ 3	1 - (-2) $\implies$ 3	4 $\implies$ 4	1 $\implies$ 1
sum	0 $\rightarrow$ 1	1 $\rightarrow$ -1	0 $\rightarrow$ 3	3 $\rightarrow$ 6	0 $\rightarrow$ 4	4 $\rightarrow$ 5
max	1	1	3	6	6	6

Therefore giving us a maximum size sub-rectangle of value = 6.