(3) **(2-D maximum-sum subarray)** (30 points) In the 2-D Maximum-Sum Subarray Problem, you are given a two-dimensional $m \times n$ array $A[1:m, 1:n]$ of positive and negative numbers, and you are asked to find a subarray $A[a:b, c:d]$, where $1 \le a \le b \le m$ and $1 \le c \le d \le n$, such that the sum of its elements, $\sum_{a \le i \le b} \sum_{c \le j \le d} A[i,j]$, is maximum.

(a) (30 points) Using exhaustive search, design an algorithm that runs in $O(m^2 n^2)$ time using $O(\min\{m,n\})$ working space.

The *working space* of an algorithm is the memory it uses beyond what is needed to store the input.

(Hint: First design a straightforward algorithm that achieves $O(m^3 n^3)$ time, and then optimize it.)

(b) **(bonus)** (10 points) Using divide-and-conquer, design an algorithm that runs in $O(m^2 n \log n)$ time using $O(n)$ working space, by dividing the array vertically.

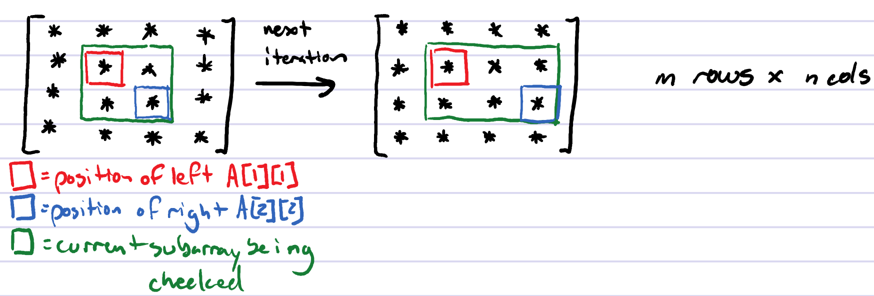
(Hint: You may need to know that the recurrence

$$T(m,n) \;=\; 2T(m,n/2) \;+\; O(m^2 n)$$

has the solution $T(m,n) = O(m^2 n \log n)$.)

a) In order to find the max.-sum subarray we could iterate over all possible subarrays contained in the original array. We can do this by having a starting position, lets call it left, that begins in the top left corner of the array and traverses to every array position until it reaches the bottom right. For each position of left we would have another position holder, lets call it right, that begins where left is and can't have any row indexes less than left's row positions. Right will also traverse through ever position of the array equal to or to the right of and equal to or below the index of left. With this we will find every subarray contained within the array

ex.



Next we have to have a way to calculate the sum of the subarray and keep track of that position if it currently holds the highest value found so far. We can do so by keeping a running total of each element explored in the subarray and another variable that keeps track of the max. sum found so far. Once a new sum is encountered that is larger than the current max then the max becomes the sum and we save the positions of the subarray

```
M:=0   -> keep track of max sum found  }  extra space O(1)
S:=0   -> running sum total            }
for LeftRow:=0 to m begin                O(m)
   for LeftCol:=0 to n begin             O(n)
      S=0
      for RightRow:=0 to m begin         O(m)
         for RightCol:= 0 to n begin     O(n)
            S= S+ A[RR][RC]
            M=S
```

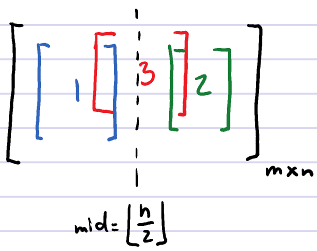
a = LeftRow
b = RightRow
c = LeftCol
d = RightCol

} extra space $O(1)$

end
end
end
end

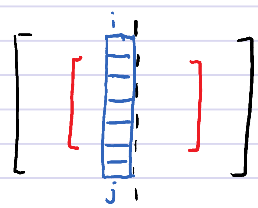
run-time = $O(m^2 n^2)$ extraspace = $O(1)$

b) First we would want to split the array in half and if the solution is in either the left half or the right half then we can find those by recursively calling the function with new bounds equal to the new halves



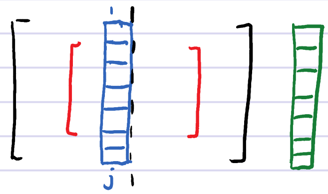
case 1 - solution is in left half
recursive call from 0 to mid
case 2 - solution is in right half
recursive call from mid to n
case 3 - solution spans both halves
we can find the best set of columns on both sides that join to the halfway point so when added together they produce an even greater sum

To find the max-sum subarrays for each side we must iterate through each column



We then keep an auxilary array that maintains the running total of sums for each row of the left and right half of the 2D-array

Going through the elements of a column is $O(m)$

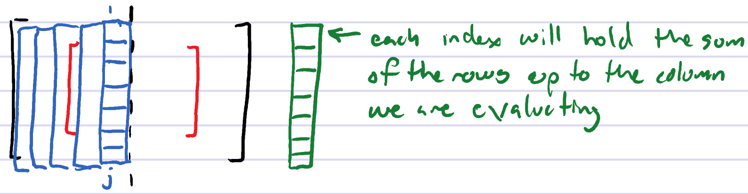


for the first iteration this array stores the values of the column up against the halfway point

of the auxillary array

Once the running sum for the columns is filled we can find the max sum subarray using dynamic programming as we did in class which is $O(n)$, but in this case is on the columns so it would be $O(m)$. Anytime a new max value is found we can update a variable holding that value.
Once the next column is introduced we add the row to the current running row totals and update the running sum for that row with that value

Iterating over all columns will be in O(n)



When it is done going through every column on the left side we have a max value for the left and we can run the algorithm on the right side, but in the other direction.
Once both sides are done and a max from both the left and right recursive calls has been determined and then the max between that and the sum of left and right halves that span the middle is found we have found the subarray with the max-sum.

This works because we split the problem into three sections (left, right, middle). The left and right subprolems are solved by recursively calling the function and setting a new middle. The middle max-sum subarray is found maintaining a running sum of the row values and only updating the max value if we find a subarray in the auxilary array that is greater than the current max.

Time analysis

$2T(m, n/2)$ { $T(m, n/2)$ recursive call to left half
$T(m, n/2)$ recursive call to right half
determine the max of the the two recursive calls
find mid point
Determine max-sum subarray of left half that meets the halfway point
$O(m^2 n)$ { $O(m)$ - iterate through all elements of the column
$O(m)$ - find max-sum subarray of running-sum auxilary array
$O(n)$ - go through all columns
$O(m^2 n)$ Do the same as above for the right half
$O(1)$ Add the max of the left and right halves and compare to current max

The total time is $2T(m, n/2) + O(m^2 n)$
Extra space is the amount of space taken up by the auxilary array that holds the running sums of the columns which would be equal to the size of the columns → $O(n)$