Start at [0][0]

-1	7	-8	10	-5
-4	-9	8	-6	0
5	-2	-6	-6	7
-7	4	7	-3	-3
7	1	-6	4	-9

Ans
15
6
(2,0)
(2,1)
(3,1)
(3,2)
(3,3)
(4,3)

```
int check = 0;
int sum = 0;
int count = 0;
sum = func(*matrix[0][0], *check)
int func( m[][], c)
    c = c + m[i][j]
    if(c ≤ 0)                    <  if(count ≥ 1)
        count = 0                    return max(c, max(func[i+1][j], func[i][j+
        c = 0
        return max(func([i+1][j], c), func([i][j+1], c))
    if(check > 0)
        count += 1
        return max(func([i+1][j], c), func([i][j+1], c))
```

if { m[i][j] ≥ 0 i < rows; j < cols sum += max(m[i][j] + m[i+1][j], m[i][j] + m[i][j+1], m[i+1][j], m[i][j+1])

m[i][j] < 0 " sum = max(m[i+1][j], m[i][j+1])

i > rows or j > cols return check

This algorithm has $\Theta(nm)$ runtime for a $n \times m$ matrix, since all the checks are $\Theta(1)$ for all elements of the matrix.

```
int rows = 0;
int cols = 0;
FILE *ex;
ex = fopen("EX1.txt", "r");
fscanf(ex, "%d ", &rows);
fscanf(ex, "%d ", &cols);

int matrix[rows][cols];

for(int i=0; i<rows; i++){
    for(int j=0; j<cols; j++){
        fscanf(ex, "%d ", &matrix[i][j]);
    }
}
int sum = 0;
int count = 0;
sum = findSum(matrix[0][0], count, 0)

int findSum(m[i][j], cnt, check){
    if(i>rows || j>cols){
        return check; }
    check = check + matrix[i][j];
    if(check <= 0){
        check = check - matrix[i][j];
        if(cnt >0){
            return max(check, findsum[i+1][j], findsum[i][j+1]
        return max(findsum[i+1][j], findsum[i][j+1])
    if(check >0){
        return max(findSum[i+1][j], check), findSum[i][j+1], check)
```