pragy@scaler.com
7351769231

Sum of all submatrices → 200/100/200 TLE

↳ $O(n^3)$ Brute

↳ $O(n^2)$ TC + $O(n)$ SC Prefix Sums array

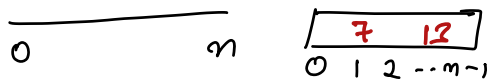
↳ $O(n^2)$ TC + $O(1)$ SC carry forward.

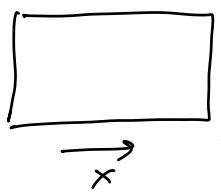
↳ $O(n)$ $O(1)$ space

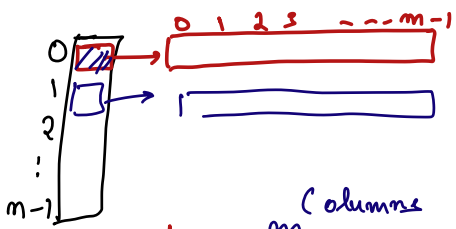
time

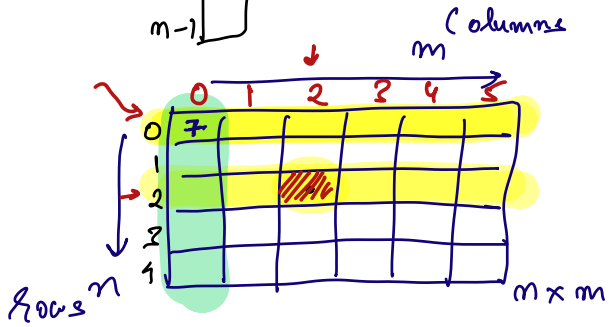
2D Matrices → List of Lists

Arrays → 1 dimensional.









A =

A[0][0]

A[0]

A[2][2]

A x no syntax

numpy

A[:, 0]

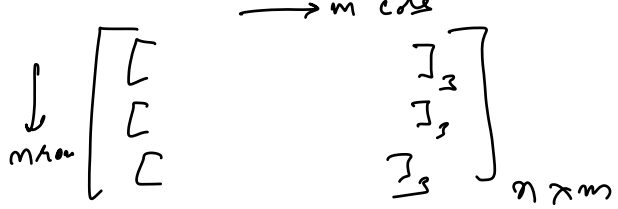
library

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Rightarrow \begin{matrix} 0 \\ 1 \\ 2 \end{matrix}\begin{bmatrix} [1,2,3] \\ [4,5,6], \\ [7,8,9], \end{bmatrix}$$

← Lists

$M = [[1,2,3], [4,5,6], [7,8,9]]$

In Arrays → homogeneous



Given a Matrix M, row number R → 1-indexed
↳ print the R^{th} row of M.

$M[R-1]$ → 0-indexed

$A_1 = [3\ 5\ 7\ 0\ 8]$ (indices 0 1 2 3 4)

$i = 1$ second

$A[i]$

A, i^{th} element from the array → $A[i-1]$

$i = 1$ → first element

$M_{n \times m}$
R

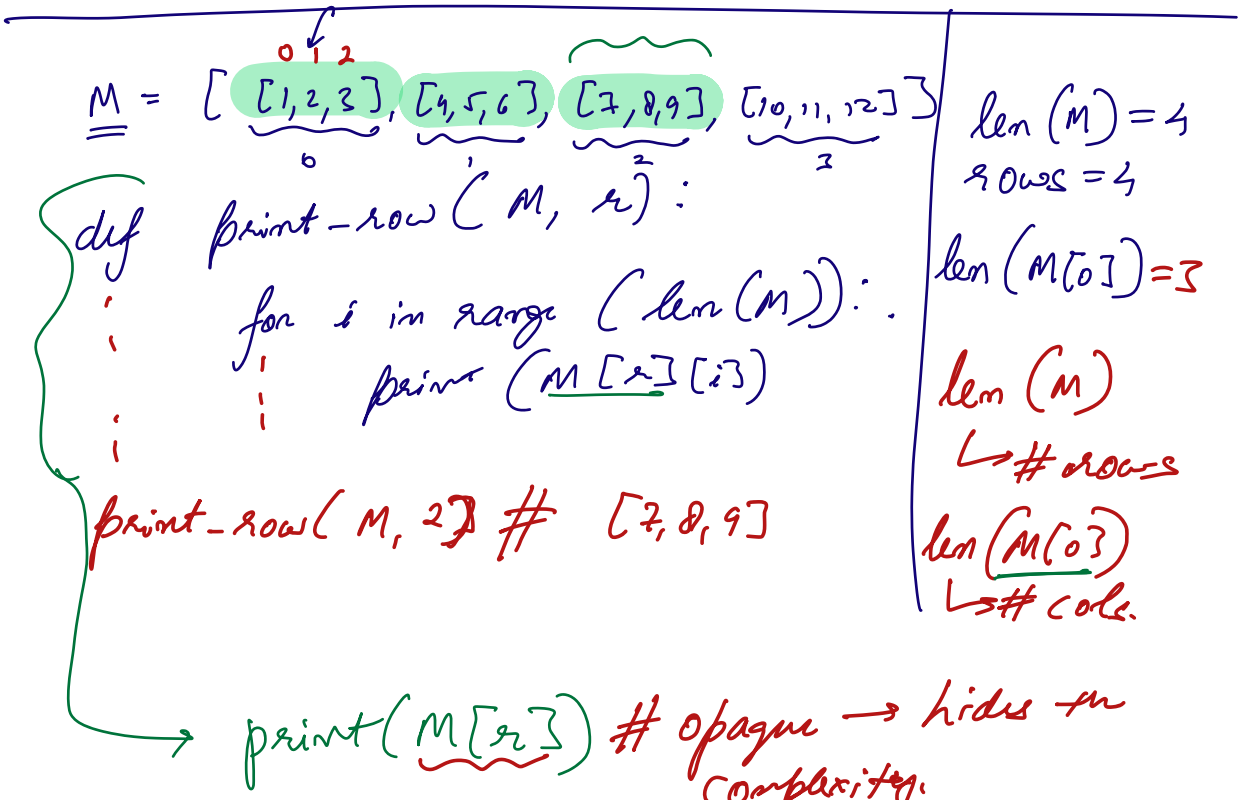
print($M[R]$) → Time complexity? $O(m)$ time

opaque, hides something

$M[R]$ → index operation → $O(1)$ time

M

```
for i in range(len(M[R])):
    print(M[R][i])
```


M = [[1,2,3], [4,5,6], [7,8,9], [10,11,12]]
0 1 2
0
1
2
3
def print_row(M, r):
for i in range(len(M)):
print(M[r][i])
print_row(M, 2) # [7,8,9]
len(M) = 4
rows = 4
len(M[0]) = 3
len(M)
↳ # rows
len(M[0])
↳ # cols.
print(M[r]) # opaque → hides the complexity.
len(____) → how much time does this take?
↳ O(1) time mostly
↳ generator → O(n) time
↳ overridden to __len__ → arbitrary
n = len(some_list) # O(1)
Data Structure internally keeps track of # of element it contains
0
1
M = [[1,2,3], [4,5,6]]
0 1 2
0 1 2
len(M) 2
len(M[0]) 3

len(M[1]) 3

len(M[2]) Index Error : 2

---

given a 2D Matrix M.

return an array containing the rowwise sums

inp.

M = [[0,3,8], [2,7,9], [1,5,-6], [3,8,9]]

output?

[11, 18, 0, 20]

```
class Solution:
    def solve(self, M):          # M: Matrix
        row_sums = []            # empty list
        n = len(M)               O(1)
        m = len(M[0])            O(1)
        for row in range(n):     => O(n)
            sum = 0
            for col in range(m): => O(m)
                sum += M[row][col]   O(1)
            row_sums.append(sum)     O(1)
        return row_sums
```

TC: O(n·m)

~~O(n²)~~

SC: O(n)

~~O(m)~~

---

l = [1,2,3]

l.append(10) # l = [1,2,3,10]

```
class Solution:
    def solve(self, M):
        return [sum(row) for row in M]
```

list comprehension.

for row → don't count in SC

M = [[1, 2, 3], [4, 5, 6]] 2 x 3

```
def solve(self, M):
    row_sums = []                 # m items
    n = len(M)          → 1
    m = len(M[0])       → 1
    for row in range(n):          # 1
        sum = 0                   # 1
        for col in range(m):
            sum += M[row][col]
        row_sums.append(sum)
    return row_sums
```

O(n+3) space

O(n) space

```
A = [ - - - ]
for i in range(len(A)):
```

executed only once

---

given Matrix M

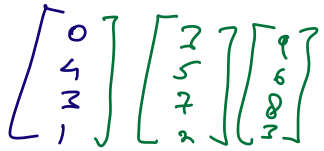
return columnwise sum

i/p

M = [[0, 3, 9], [1, 5, 6], [3, 7, 8], [1, 2, 3]] 4 x 3

cols

o/p = [8, 17, 26]

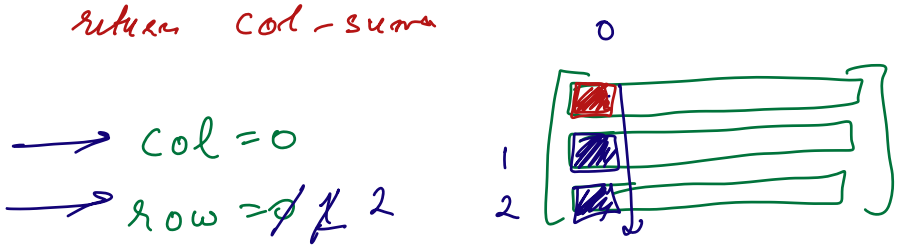


```
def solve(M):
    col_sum = []
    n = len(M)
    m = len(M[0])
    for col in range(m):
        sum = 0
```

TC O(n.m)

SC O(m)

```
    for row in range(m):
        sum += M[row][col]
    col_sums.append(sum)
return col_sums
```



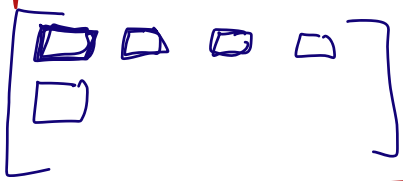
col = 0

row = ~~0~~ ~~1~~ 2

sum = 0

Col sums while iterating over rows first

```
col_sums = [0, 0, 0, 0, ...]   O(m) space
n = len(M)
m = len(M[0])
for row in range(n):           O(n)
    for col in range(m):       O(m)
        col_sums[col] += M[row][col]
return col_sums
```



avoid.

```
l = [0]*m
```

```
l = []
for _ in range(m):
    l.append(0)
```

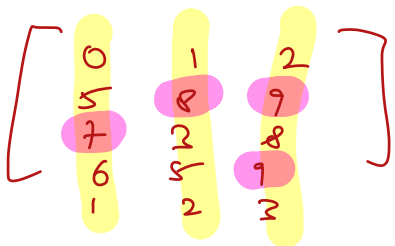
```
l = [0 for _ in range(m)]
l = M[0][:]
l = deepcopy(M[0])
```

Find the columnwise Max

i/p

$M = [[0,1,2], [5,8,9], [7,3,8], [6,5,9], [1,2,3]]$

o/p = [7, 8, 9]



TC / SC

$O(n \cdot m)$ ~~$O(n)$~~ $O(m)$

---

given Two matrices, M1, M2

return a new matrix which is the sum of M1 & M2

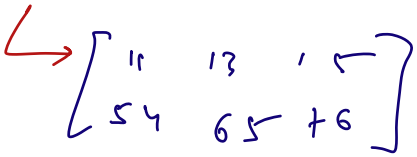
i/p

M1 = [[1,2,3], [4,5,6]]

M2 = [[10,11,12], [50,60,70]]

o/p

= [[11, 13, 15], [54, 65, 76]]



$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = [\times]$

→ not defined for matrices with diff dimensions.

```
n = len(M1);  m = len(M1[0])        → list comprehension
result = [[0]*m for _ in range(n)]
for i in range(n):
    for j in range(m):
        result[i][j] = M1[i][j] + M2[i][j]
return result
```

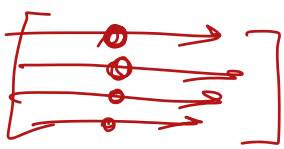
→ O(n·m) TC

O(n·m) SC

```
result = [[0]*m] * N
```

gives you a Matrix of size N x m

copies by reference



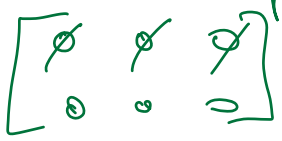
```
result = [[0]*m for _ in range(n)]
```

~~result~~ = [ ] [ ] X

= [[ ]] → [ ]

---

create & replace



vs

[ ]

[[ ]]

[[0]]

[[0 | 3]]

insert

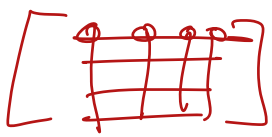
```
def solve (M1, M2):
    n = len(M1)
    m = len(M1[0])
    result = [] # dimension?

    for row in range (n):
        result_row = [
        for col in range (m):
            result_row.append(M1[row][col] + M2[row][col])
        result.append(result_row)
    return result.
```

Dry Run:

$$M1 = \begin{matrix} & 0 & 1 & 2 \\ 0 & [1 & 2 & 3] \\ 1 & [4 & 5 & 6] \end{matrix}$$

$$M2 = \begin{matrix} & 0 & 1 & 2 \\ 0 & [10 & 20 & 30] \\ 1 & [40 & 50 & 60] \end{matrix}$$

```
def solve (M1, M2):
    n = len(M1)        ——— n = 2
    m = len(M1[0])     ——— m = 3
O(n.m) ← result = [] # dimension? → result = [ [11 22 33], [44 55 66] ]
    for row in range (n): ←
        result_row = [] ←
        for col in range (m):
            result_row.append(M1[row][col] + M2[row][col])
        result.append(result_row)
    return result.
```



TC $O(n \cdot m)$ SC $O(n \cdot m)$

10:45 → 10:55

doubts

```
n = len(A)          # O(1)
for i in range(len(A)):      } O(n)
```

(len(A) → O(1))

=

```
m = max(A)          O(n)
for i in range(max(A)):      } O(max)
```

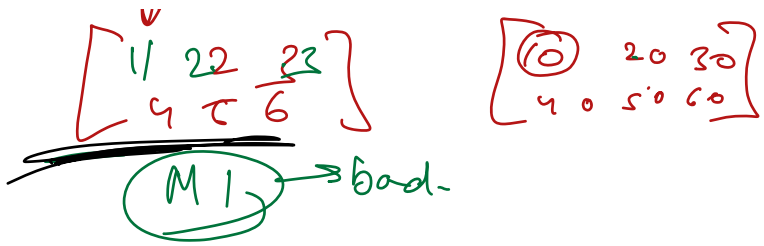
(max(A) → run only 1 time)

=

A = [1, 2, 3]

m = max(A)

```
add(M1, M2):
    result = M1          # not a copy ← modifying your input
    for ____
        for ____
            res[i][j] = M1[i][j] + M2[i][j]
return result
```



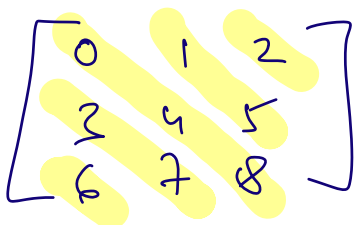
M. copy → bad →

dup copy

---

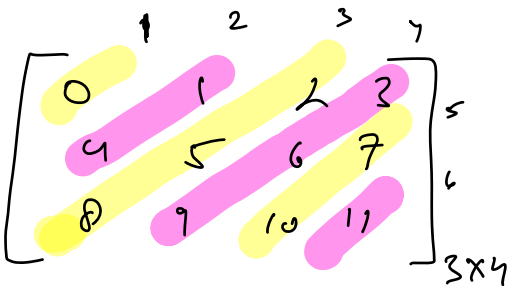
given Matrix of n x m

print the diagonals of Matrix

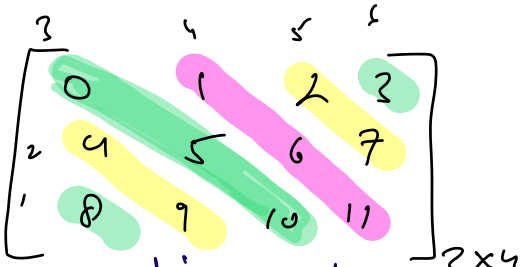


left - right → diagonals

anti-diagonals → R → L



anti-diagonals

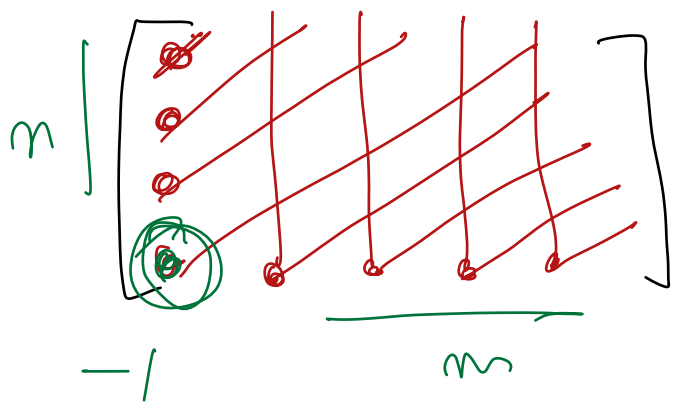


diagonals

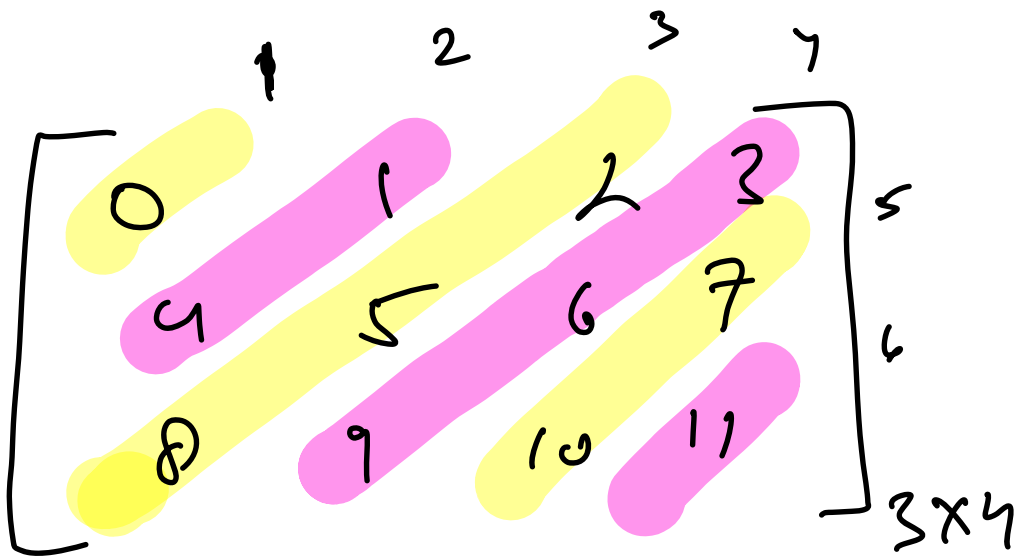
Matrix $n \times m$

# of diagonals $= n + m - 1$

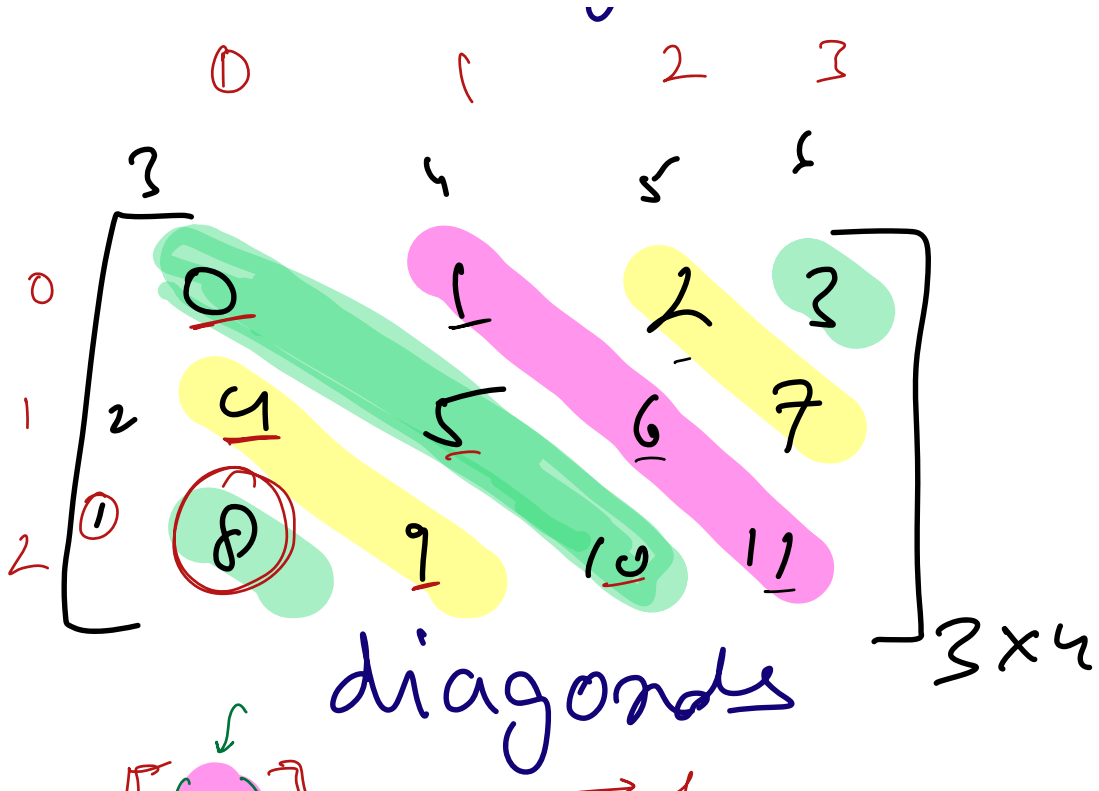
# of anti-diagonals $= n + m - 1$



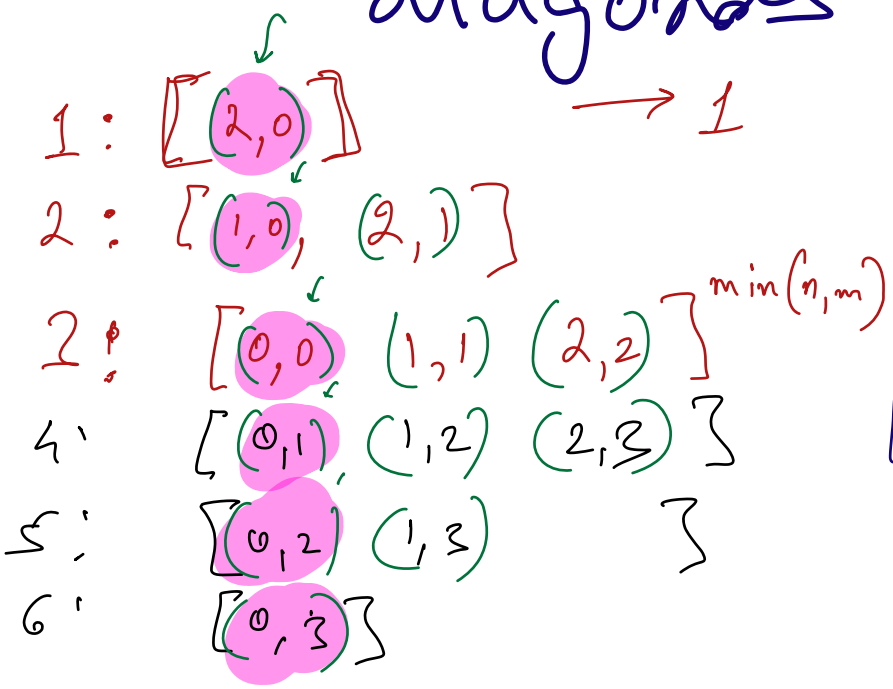
```
def diagonals(M):
```

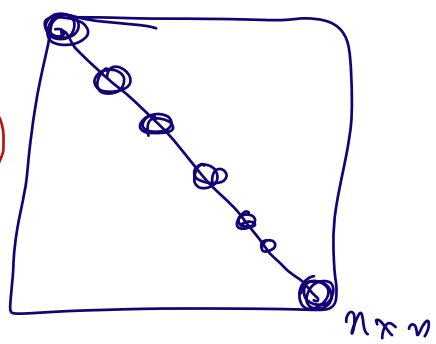


anti-diagonals

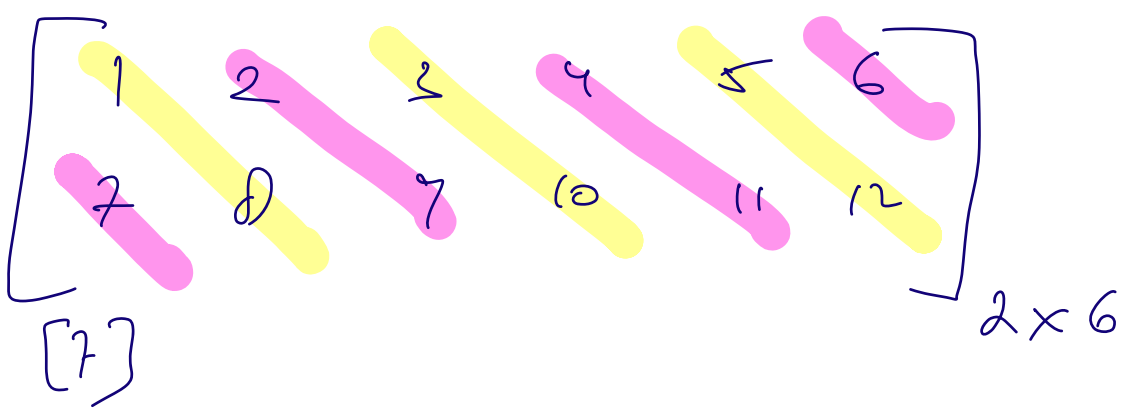


diagonals



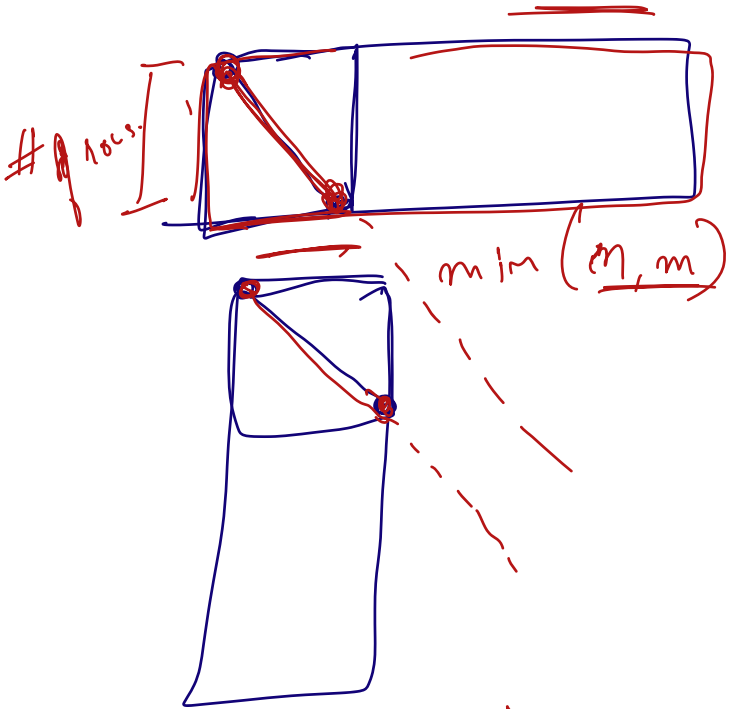


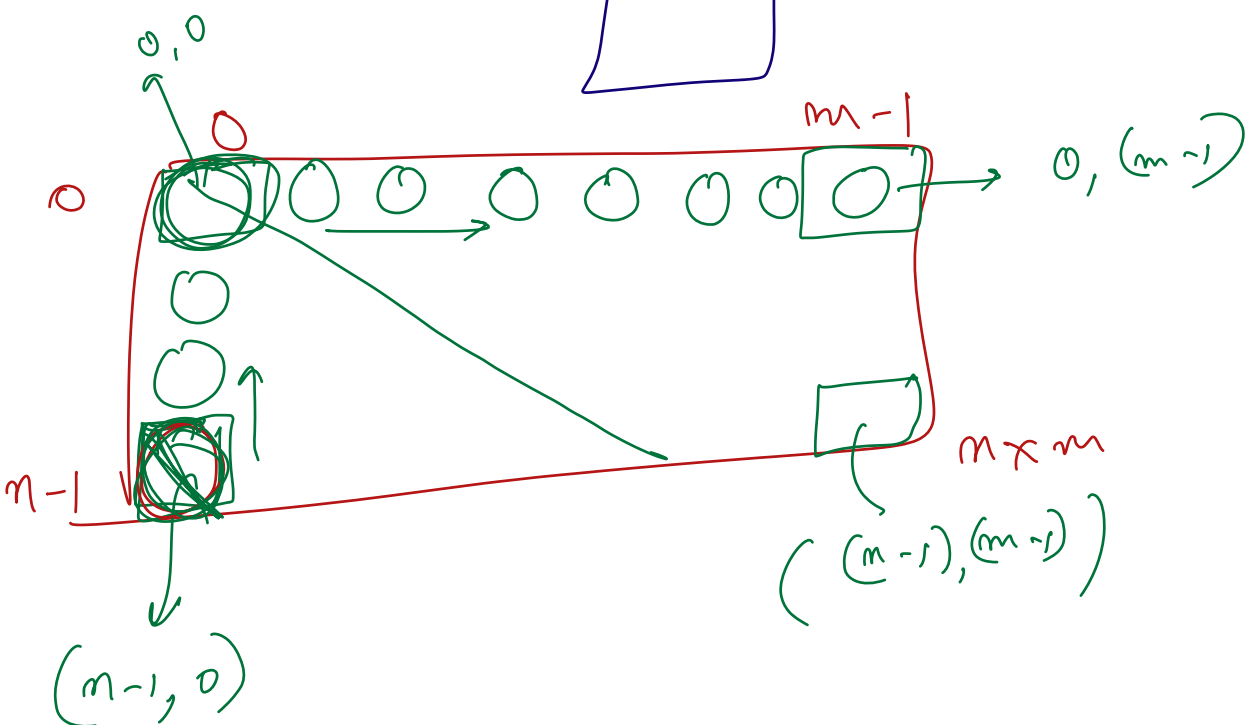
diag = n

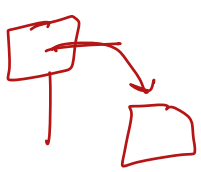


[7]

[1 8]

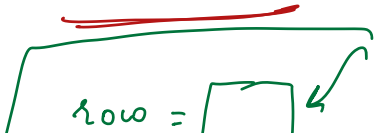






s_row = n-1

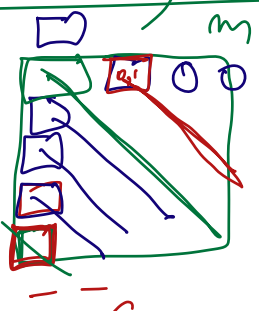
s_col = 0



```
col = [
diag = [ ]
while row < n and col < m:
    diag.append( M[row][col] )
    row += 1
    col += 1
```

$(n + m - 1)$

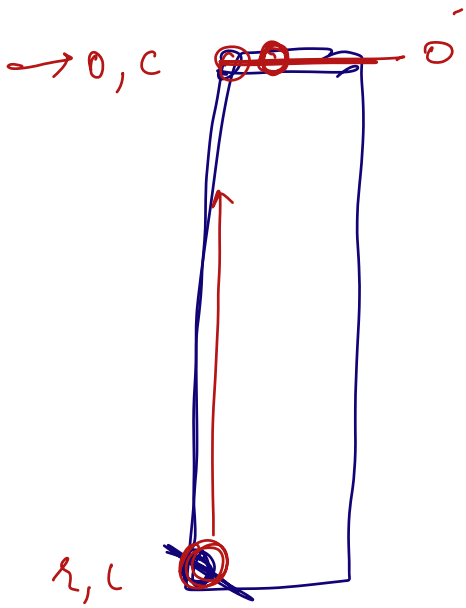
```
def print_diagonals(M):
    n = len(M)
    m = len(M[0])
    diag_start_row = n-1
    diag_start_col = 0
    # go over each diagonal
    while diag_start_col < m:
        # print the diagonal
        diag = [ ]
        row = diag_start_row
        col = diag_start_col
        while row < n and col < m:
            diag.append(M[row][col])
            row += 1
            col += 1
        print(diag)
        if diag_start_row > 0:
            diag_start_row -= 1
        else
            diag_start_col += 1
```



$TC = O((n+m-1) * \min(n,m)) = O(n \cdot m)$

$SC = O(\min(n,m))$

$\min(n,m)$



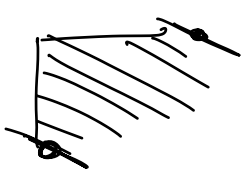
Observations:

(1) in a diagonal if the current cell is (i,j)
next cell is $(i+1, j+1)$

(2) the diag continues till we hit the boundary
$i < m$ & $j < n$

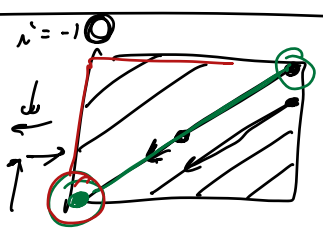
(3) start index of 1st dia = bottom-left $= ((n-1), 0)$

(4) to change the diag → first move up
→ move right

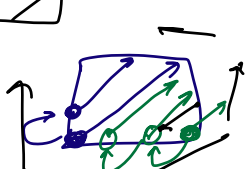


---

anti-diagonals



(1) if current cell is (i,j)
R → L next cell is $(i+1, j-1)$



↳ → R (i-1, j+1) ↙ L → R

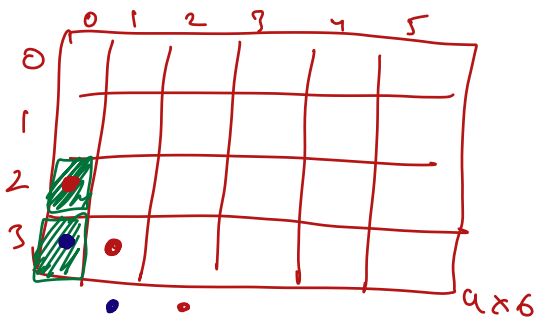
(2) diag continues while $i \geq 0$ && $j < m$

$(0 <= i < n$ and $0 <= j < m)$

(3) we start from bottom right corner $(n-1, m-1)$

(4) first go left ~~go~~ then go up

```
def print_diagonals(M):
    n = len(M)          → 4
    m = len(M[0])       → 6
    diag_start_row = n-1  → ~~1~~ 2
    diag_start_col = 0    → 0

    while diag_start_col < m:
        diag = []
        row = diag_start_row
        col = diag_start_col
        while row < n and col < m:
            diag.append(M[row][col])
            row += 1
            col += 1
        print(diag)
        if diag_start_row > 0:
            diag_start_row -= 1
        else:
            diag_start_col += 1
```

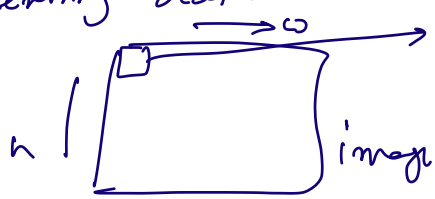


[•]

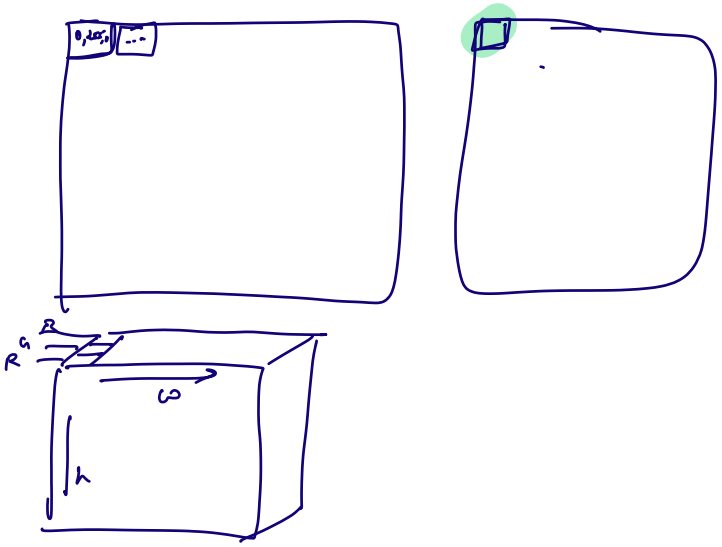
[• •]

---

representing data



pixels → R, G, B — 0-255 0-255 0-255

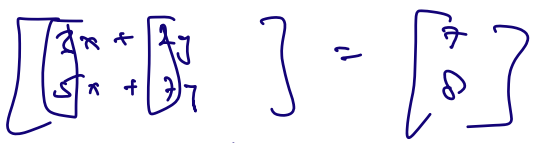


---

image → 3d data

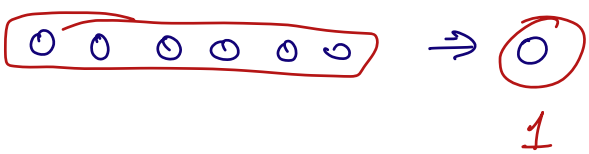
[images] → 4d data

2-d data



$$\begin{bmatrix} 3 & 2 \\ 5 & 7 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad \begin{bmatrix} 7 \\ 8 \end{bmatrix}$$

Starting index being 0 is just a convention.

↳ binary → 

1

Matlab → indices start from 1

R →

```
M = [ ]
row = [0]*m
for i in range(n):
    M.append(row.copy())
```

```
M = [[0]*m for _ in range(n)]
```

`M = [ [0], [1,2], [3,4,5] ]` → is this valid.

lists → yes → valid in Python.

arrays strictly.

`M[3][2]` → array of size 3 in which each element is itself an array of size 2

`len(____)` → usually an O(1) operation

but no guarantee

`len(list)` O(1) always

`len(tuple)`

`len(str)`

x = M[i] ← indexing is O(1)

↳ is an array

---

1. primer content of python
2. next week → special python

generators → memory efficient way of working with lists

```
l = [0 --- 1 billion]
for value in l:
    print(value)
```

each int → 4 bytes

4 GB

O(4 GB)

```
i = 0
while i <= 100 --
    print(i)
    i ++
```

space is O(1)

yield operator

lazy evaluation.

0 — 1 billion

```
for value in range(1bil):
    print(value)
```

```
class MyCustomList:
    def __init__(self, *args, **kwargs):
        self.items = list(*args, **kwargs)
        self.count = 0

    def __len__(self):
        return self.count

    def append(self):
        self.count += 1

l = MyCustomList()
len(l)
```

magic / dunder methods
operator overloading

O(1)

---

data - augmentation

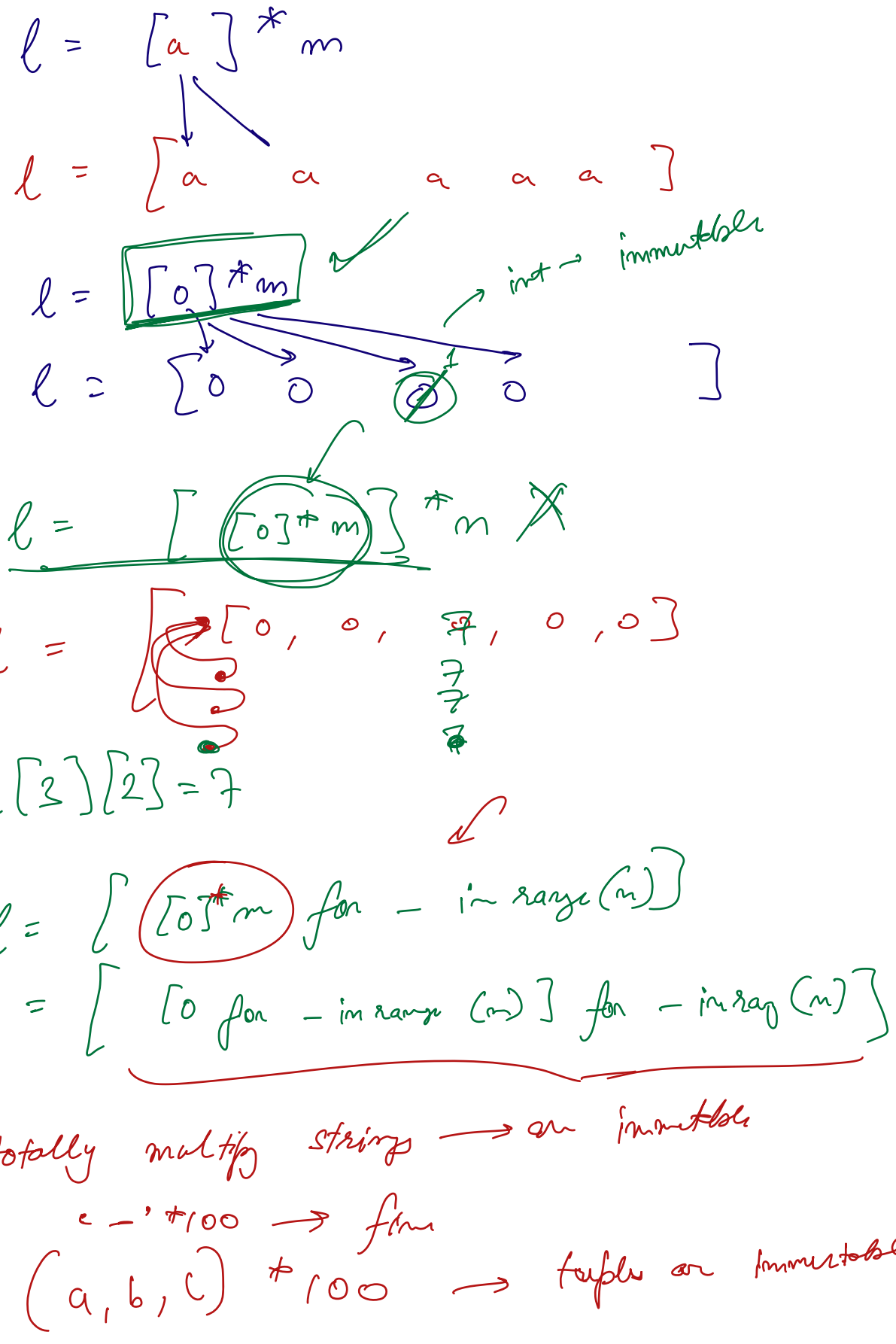
ar / lists → len → O(1)

max → O(n)

---

np.array → homogenous

l = [ ] → heterogenous


l = [a] * m
l = [a a a a a]
l = [0] * m
int → immutable
l = [0 0 7 0]
l = [[0] * m] * n
l = [0, 0, 7, 0, 0]
7
7
7
l[3][2] = 7
l = [[0] * m for _ in range(n)]
= [[0 for _ in range(m)] for _ in range(n)]
totally multiply strings → an immutable
e.g. 'a' * 100 → fine
(a, b, c) * 100 → tuple an immutable

P1 → $O(n^2)$

P1 → $O(n)$
$O(1)$

NP - complete

---

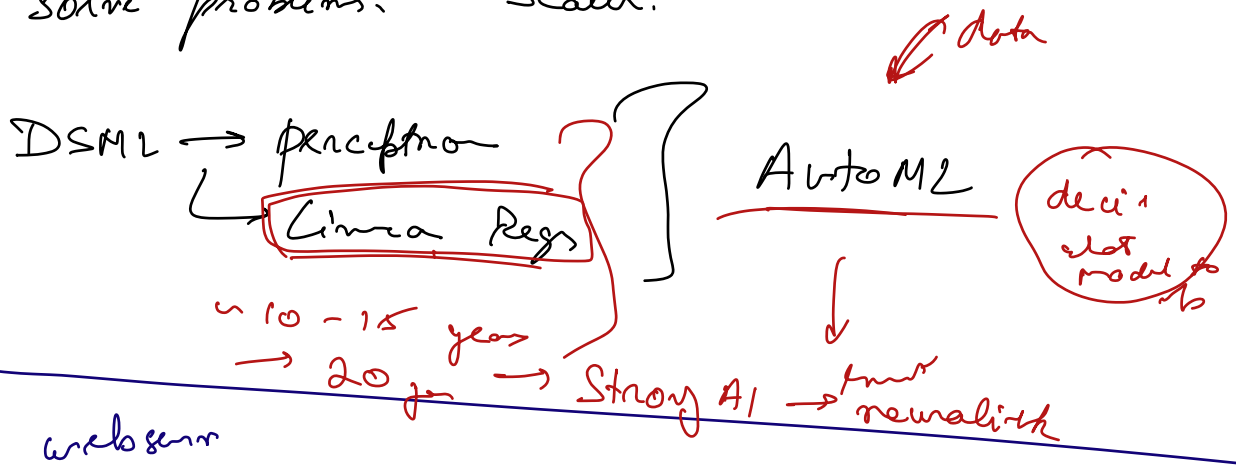
a = [1, 2, 3]
b = [1, 2, 3]

---

id (___) → memory address.

---

solve problems. ← Scalar.



---

webserver
↳ server
↳ database
↳ api
↳ auth
→ auth

→ boilerplate

supabase.io.

→ low code platform
no - code

7351769231

pragy@scaler.com.