"we do not sit around and wait for other people, we just make, and we do"

Content:

01. Prefix Sum in matrix
02. Rain water trapped SC: O(1)
03. Minimum Swaps
04. Insert intervals

01. Prefix Sum in matrix:

Q Given a matrix of size N * M & Q queries

For each query, find sum of given submatrix

Note: TL is top left & BR is bottom right.

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

Queries = 2

	(T,L)	(B,R)
01.	(2,1)	(4,2)
02.	(1,1)	(3,3)

(2,1) (4,2) => 9 + 8 − 1 + 2 + 2 + 6 = 26
(16) (10)

(1,1) (3,3) => 10 + 19 + 4 => 33

BF: TC: $(Q \times N \times M)$ SC: $O(1)$

1. for each Query $O(Q)$
2. (2, 1) (4, 2)
   i j i j

```
sum = 0                                   2, 1
for i in range(2, 4+1)                    2, 2
    for j in range(1, 2+1)
        sum1 += A[i][j]
ans.append(sum)
```

At worst case: we will visit entire matrix.

Optimized approach:

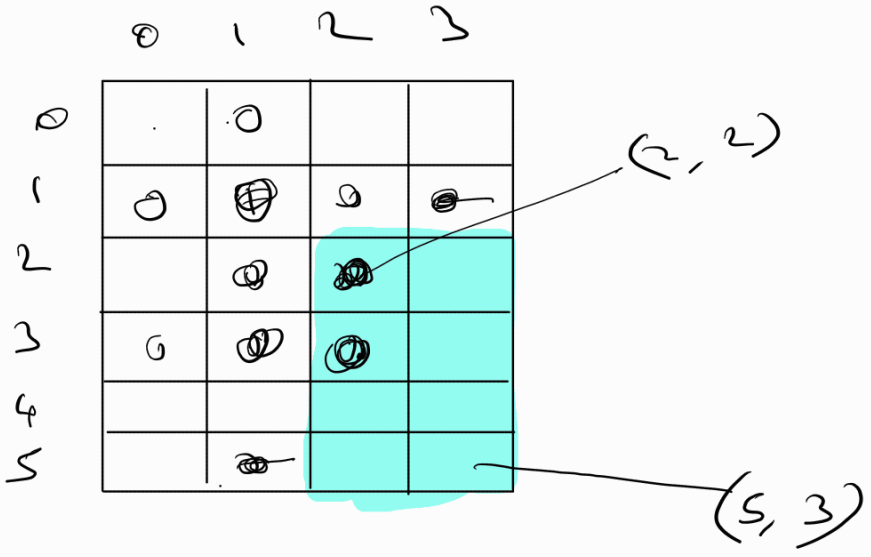
Prefix Sum:

For 1D array: [A[i], A[i] + Pf[i-1] ...]

↳ Sum of all elements from $0 \rightarrow N-1$

For 2D array:

Sum of all elements from (0,0) to (n-1, m-1)



Query: (2,2) (5,4)

$= Pf(5,4) - Pf(5,1) - Pf(1,4) + Pf(1,1)$

Query (1,2) ↔ (4,3)

$= Pf(4,3) - Pf(4,1) - Pf(0,3) + Pf(0,1)$

end $(4, 2-1)$ $((1-1), 3)$ $((1-1), (2-1))$

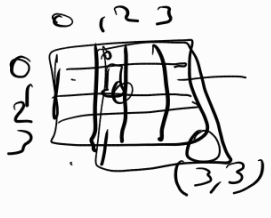
(1,2) (3,3)

$= Pf(3,3) - Pf(3,1) - Pf(0,3) + Pf(0,1)$

$3, (2-1)$ $((1-1), 3)$

Generalizing it!

$R = [(a_1, b_1), (a_2, b_2)]$ (n, m)



```
Sum = pf(a2, b2)
if (b1 > 0):
    Sum -= pf(a2, b1 - 1)
if (a1 > 0)
    Sum -= pf(a1 - 1, b2)
if (a1 > 0 and b1 > 0)
    Sum += pf((a1 - 1), (b1 - 1))
```

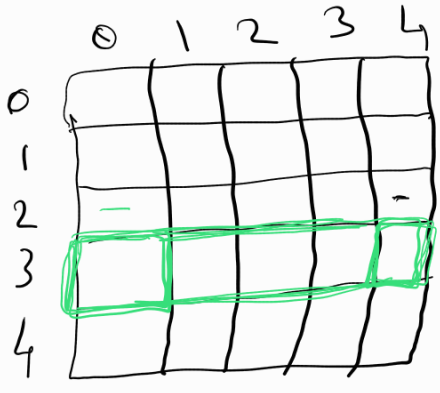
[a1, b1
(3, 0)
(3, 4)]
a2 b2

Sum = pf(3, 4)

~~Sum -= pf(3, 0-0)~~

Sum -= pf(3-1, 4)

~~Sum += pf(3-1, 0-0)~~



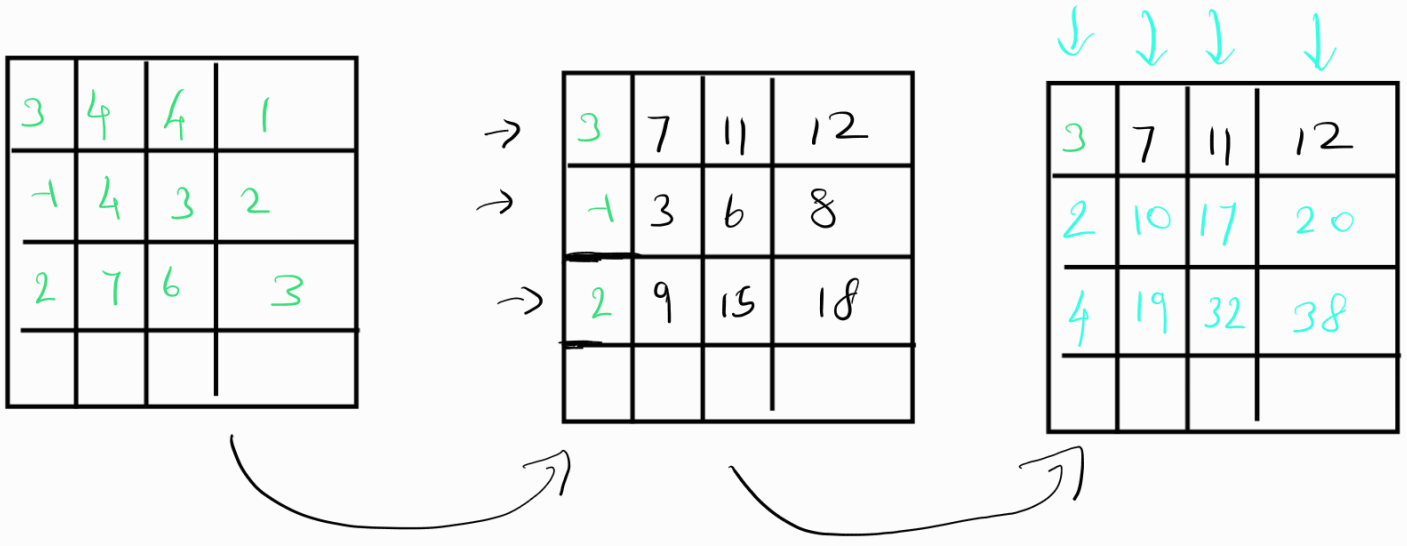
Next how to build prefixSum for matrix

1. Apply sum in row wise
2. apply sum in column wise

Initialy

Apply row wise

Apply Column wise



TC: $O((N \times M) + (N \times M))$

$\Rightarrow O(N \times M)$

SC: $O(N \times M)$

To apply and for each Query $O(Q)$

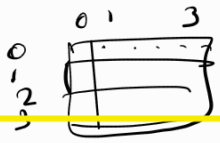
overal TC: $O((N \times M) + Q)$

SC: $O(N \times M)$

steps:

1. build prefix Sum matrix, with 2 loops.
2. for each Query Apply the formula and access the interval sum.

Code:



To have A as prefix Sum array

```
N = len(A)
M = len(A[0])

for i in range(N)  # rows
    rowSum = 0
    for j in range(M)  # columns
        rowSum += A[i][j]
        A[i][j] = rowSum
```

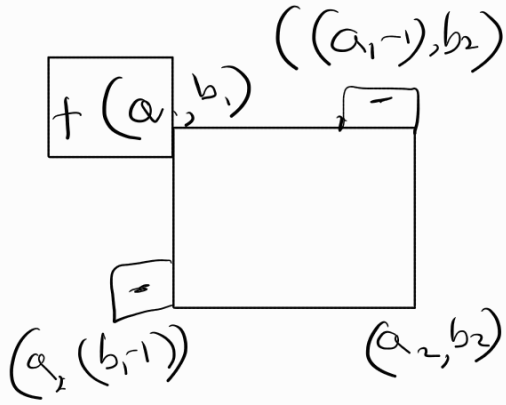
To have separate prefix Sum array

```
PfSumMat = [ ]

for i in rang(N):
    rowSum = 0
    temp = [0] * M
    for j in range(M):
        rowSum += A[i][j]
        # temp.append(rowSum)
        temp[j] = rowSum
    PfSumMat[i] = temp
```

```
for j in range(M):
    colSum = 0
    for i in range(N):
        colSum += A[i][j]
        A[i][j] = colSum
```

for each query access sum from prefixSum




```
total = 0
for q in range(len(Q))
    a1, b1 = Q[q]
    b1, b2 = Q[q]
    qSum = A[a2][b2]
    if (a1 > 0 and b1 > 0):
        qSum += A[a1-1][b1-1]
    if (a1 > 0):
        qSum -= A[a1-1][b2]
    if (b1 > 0):
        qSum -= A[a2][b1-1]
```

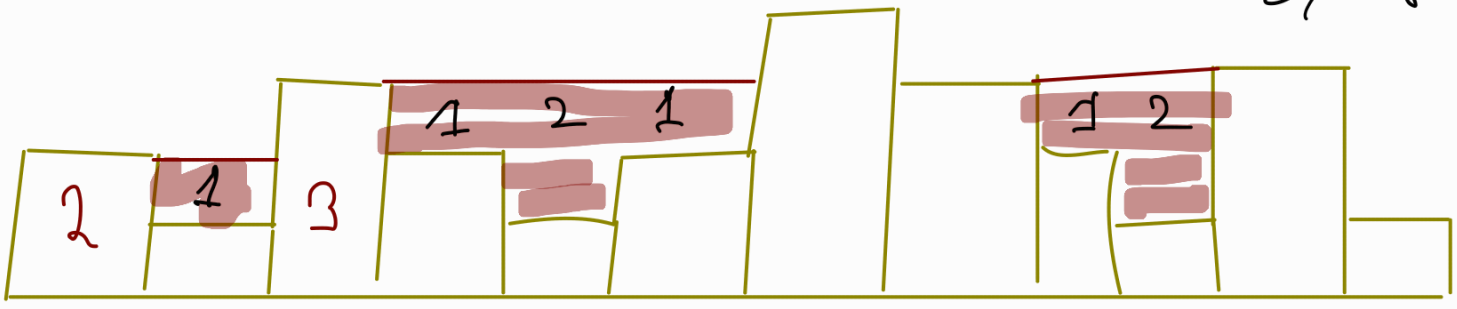
Question:

⇒ given N buildings with height of each building

⇒ Find the rain water trapped between the buildings

A = [2, 1, 3, 2, 1, 2, 4, 3, 2, 1, 3, 1]

O/P = 8



water Trapped on ith building:

min(max_left, max_right) - A[i]

Note: max_left, max_right } are including i value

Brute force Approach:

Total = 0
iterate over array O(N)
for every element from 1 to N-2
check max_ele_left O(N)
check max_ele_right O(N)
calculate Trapped water
add it to sum
return Total

TC: $O(N^2)$ SC: $O(1)$

Code:

```
A = [...], n = len(A), total = 0
# iterate over array A
for i in range(1, n-1):
    # Find max element in LHS of current element.
    k = i-1, max_left = 0
    while (k >= 0):
        max_left = max(max_left, A[k])
        k -= 1
    # Find max element in RHS of current ele.
    k = i+1
    while (k < n):
        max_right = max(max_right, A[k])
        k += 1
    # calculate Trapped water.
    water = (min(max_left, max_right) - A[i])
    if water > 0: total += water.
```

Dry run:

A = [2, 1, 3, 2, 1, 2, 4, 3, 2, 1, 3, 1]
i

i = 1

max_left = 2

max_right = 3

min = 3

total = (min - A[i]) = 8

---

optimized approach: using prefixsum O(N), SC: O(N)

1. Build prefix max of arra A from left
2. Buile suffix max of array A from right
3. iterate over A anad calculate Trapped water.

Code:

```
N = len(A)
# build prefix max
prefMax = [0] * N
prefMax[0] = A[0]
for i in range(1, N):
    prefMax[i] = max(prefMax[i-1], A[i])
sufMax = [0] * N
sufMax[N-1] = A[N-1]
for i in range(N-1, 2, -1):
    sufMax[i] = max(sufMax[i+1], A[i])
total = 0
for i in range(N):
    water = (min(prefMax[i], sufMax[i]) - A[i])
    if water > 0: total += water.
return total
```

Dry run:

A = [2, 1, 3, 2, 1, 2, 4, 3, 2, 1, 3, 1]

i i

o/p = 8

prefmax = [2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4]

Sufmax = [4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 1]

i = 1

left_max = 3

right_max = 4 3

min = 3

water = (min - A[i]) 2 1 2

total = 0 1 2 4 8 6 8

---

further optimized solution? to sc: O(1)

0 1 2 3 4 5 6 7 8 9 10

h = [4, 2, 5, 7, 4, 2, 3, 6, 8, 2, 3]

1. First find max element in array h.
2. store the max element value and it's index.

   max_build = ?

   max_build_index = ?

3. from 0 → max_buid_index, right_max is "max_build"

   ↳ we will calculate the max_left on the go.

4. from (N-1) to max_build_indx,

left_max = max_buld

right_max = we will find on the go.

Dry Run:

0 1 2 3 4 5 6 7 8 9 10

h = [4, 2, 5, 7, 4, 2, 3, 6, 8, 2, 3]

i

max_build = 8

max_buld_indx = 8

O(N)

~~i = 1~~

while (i < max_build_index) i += 1 O(N)

max_left = A[0] ~~4~~ 7 max(max_left, A[i])

max_right = max_bild 8

water = min (max_left, max_right) - A[i]

min = 4

total = 0 ~~4~~ 8 ~~10~~ ~~14~~ 15

---

max_left = max_build 8 O(N)

max_right = A[N-1]

i = N - 2

while (i > max_buld_index) i -= 1

total = ~~15~~ 16

TC: O(N + N + N)

O(3N)

=> O(N)

SC: O(1)

Code:

```
# Find max building height.
max_height = A[0], N = len(A), max_ht_idx = 0
for i in range(1, N):
    if max_height < A[i]:
        max_height = A[i]
        max_ht_idx = i
# we have the righ max building height
# iterate from 0 -> max_ht_idx.
water = 0
left_max_ht = A[0]
right_max_ht = max_height
for i in range(1, right_max_ht):
    if left_max_ht < A[i]:
        left_max_ht = A[i]
    water += min(left_max_ht, right_max_ht) - A[i]
# iterate from (N-1) -> max_height_idx.
i = N-2
right_max_ht = A[N-1]
left_max_ht = max_ht
while(i > max_ht_idx):
    right_max_ht = max(right_max_ht, A[i])
    water += min(right, left_max_ht) - A[i]
    i -= 1
return water.
```

TC: O(N)

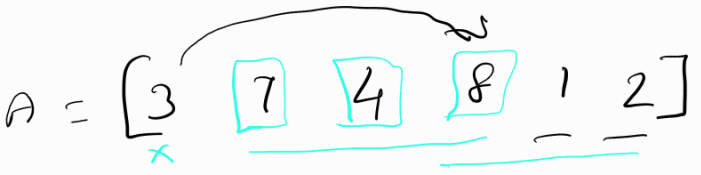
SC: O(1)

Minimum Swaps

⇒ Given int [] A and int B

⇒ Find and return minimum number of swaps required to bring all the numbers less than or equal to B together.

Note: it is possible to swap two elements not necessarily consicutives.



SwapCount = 1

ans = min(swapCount, ans)

Count = 3 = window

1. Calculate the num of elements ≤ B that we need to Bring together
2. So to bring them together our window is size "num of elements ≤ B"
3. check if elements in the window are ≤ B or not. if not increment swapCount.
4. and maintain min of swapCount and ans

Dry run: si ei

0 1 2 3 4 5

A = [3 7 4 8 1 2] B = 3

Count = 3 # window

swapCount = ~~2~~ ~~3~~ ~~2~~ 1

ans = ~~3~~ ~~2~~ 1 min(ans, swapCount)

si = 1

ei = 3 while (ei < N)

if (ei > B) swapCount += 1

if (si-1 > B) swapCount -= 1

ei += 1

si -= 1

Code:

```
# count the elements <= B
count = 0
for a in A:                          O(N)
    if a <= B:
        count += 1

# for the window of count check swapCount.
swapCount = 0
for i in range(count):               O(count)
    if A[i] > B:
        swapCount += 1
```

update ans

ans = swp count

move the window in array calculate

si = 1 , N = len(A) ① (count → N-1)

ei = count ① -(count-1)

TC: O(N)

SC: O(1)

while (ei < N):

    if (A[ei] > B) swp count += 1

    if (A[si]-1 > B) swap count -= 1

    ans = min(ans, swapcount)

    ei += 1

    si += 1

return ans.

Merge Intervals

* Merge Intervals / Insert Intervals

* Given N non-overlapping intervals sorted based on at time. Given a new interval, Merge this with existing interval if possible & return final non-overlapping interval list.

Interval = { (1,3) (4,7) (10,14) (16,19) (21,24) (27,30) (32,35) }

I_1 - (10,22)

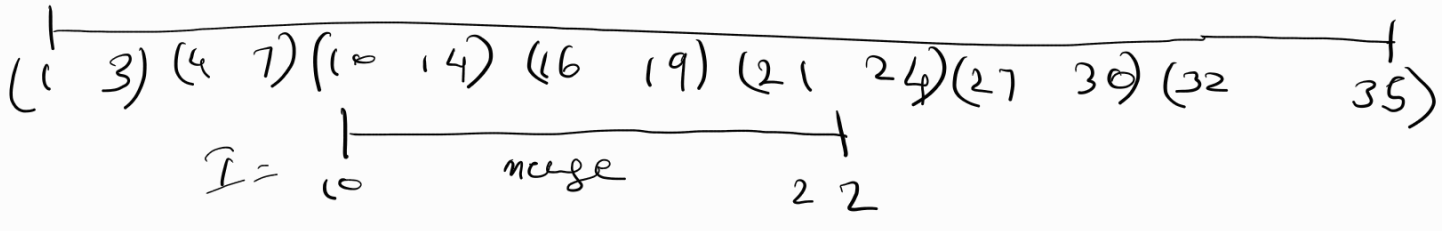
① Given n non-overlaping intervals sorted based on start time.

→ given new interval, merge this with existing intervals if possible

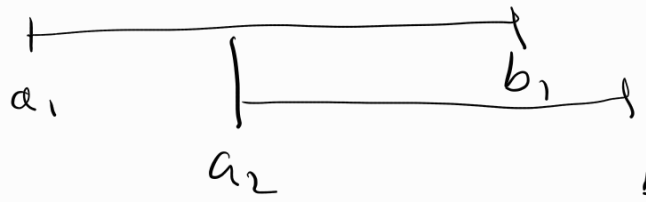
→ return final non-overlaping interval list.

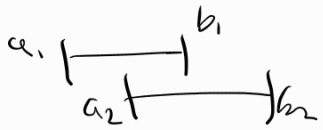
A = {(1,3) (4,7) (10,14) (16,19) (21,24) (27,30) (32,35)}

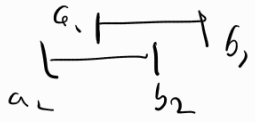
I = (10,22)



O/P = [(1 3) (4 7) (10,24) (27,30) (32,35)]



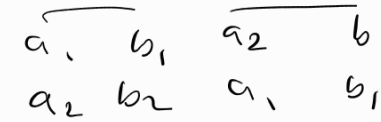




merging (a_1, b_2)

$[\min(a_1, a_2), \max(b_1, b_2)]$

(a_1, b_1) (a_2, b_2)



$(a_1 < a_2$ and $b_1 < b_2)$

(or)

$(a_1 > a_2$ and $b_1 > b_2)$

simplifying condition

$(b_1 < a_2)$

(or)

$(b_2 < a_1)$

we will che is non-overlaping condition meting or not.

if yes

NO - overlaping

if no

over lapping

merge them

```
if (e1 < s2) or (e2 < s1):
    # NO overlap
else
    # over laping
    merged = [min(s1, s2), max(e1, e2)]
```

BF:

A = {(1,3) (4,7) (10,14) (16,19) (21,24) (27,30) (32,35)}

I = (10,22)

interval1	interval2	ans
(1,3)	(10,22)	(1,3)
(4,7)	(10,22)	(4,7)
(10,14)	(10,22)	-
(16,19)	(10,22)	-
(21,24)	(10,24)	(10,24)
(27,30)		
⋮		⋮

Code:

```
N = len(A),   I = [10, 22]
for i in range(N):
    s1, e1, s2, e2 = A[i][0], A[i][1], I[0], I[1]
    if e1 < s2 ~~or e2 < s1~~
        # no overlap
        ans.append(A[i])
    elif e2 < s1:  # merge is completed.
        ans.append(interval2)
        # append remaining elements to ans
        for j in range(i, N)
            ans.append(A[j])
        return ans
```

else # overlaping.

I [0] = min(s1, s2)

I [1] = max(e1, e2)

if n-1 element also gets merged

ans.append(I)

return ans

0	1	2	3	4	5	6

A = {(1,3) (4,7) (10,14) (16,19) (21,24) | (27,30) (32,35)}

i

I = [10, 22] (10, 22)

s_2 e_2

ans = [(1,3) (4,7) (10,22) ....]

3 < 10

4 < 10

10 < 10 → 22 < 14 → merge

16 < 10 → 22 < 16 → merge

21 < 10 → 22 < 21 → merge

27 < 10 → 22 < 27 → Push I