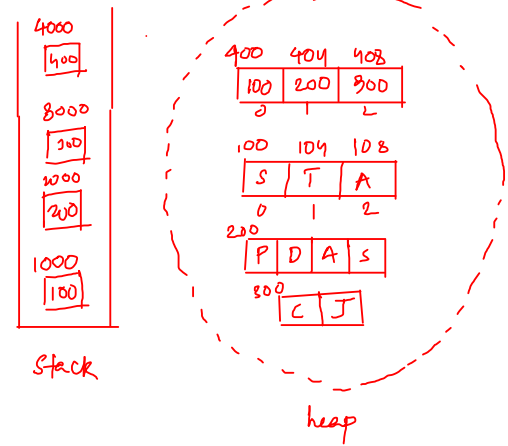
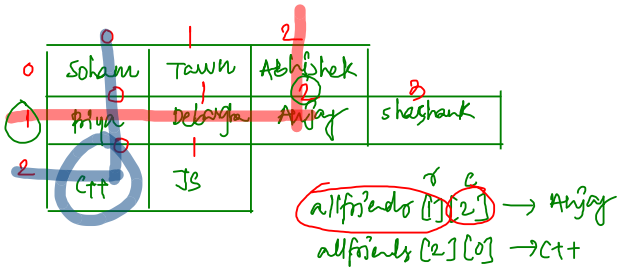
★ Array of Array (2D Array):

```
759  const friends1 = ["Soham", "Tarun", "Abhishek"];
760  const friends2 = ["Priya", "Debargha", "Anjay", "Shashank"];
761  const friends3 = ["C++", "JavaScript"];
762
763  const allFriends = [friends1, friends2, friends3];
764  console.log(allFriends);
765  console.log(allFriends[0]);
766  console.log(allFriends[1]);
767  console.log(allFriends[2]);
```

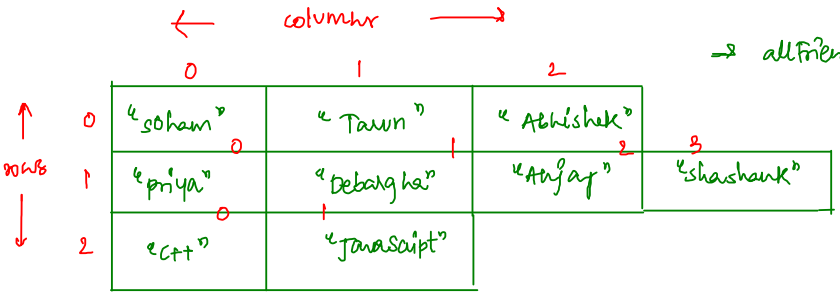


for simplicity on paper we represent them as matrices:



A	1000
f2	2000
f3	3000
alf	4000

★ Changes through A reflects in allfriends and viceversa due to the structure in heap.



→ allFriends [[ ], [ ], [ ]]
0 1 2

* When there different no. of cols in each row it is called "Jagged Array".
* In general, no. of cols will be same in every row.

Q: No. of rows

A: no. of rows = 3
= no. of smaller Arrays
= no. of elements in allFriends
= allFriends.length

Q: No. of cols

A no. of cols = length of each row
= no. of elements in each smaller array.

row = 0 ⇒ allFriends[0].length
row = 1 ⇒ allFriends[1].length
row = 2 ⇒ allFriends[2].length

rows x cols

Q: Given an N * M matrix, print it as follows

If N = 3, M = 3

	0	1	2
0	10	11	12
1	15	13	14
2	16	17	18

3 X 3

op:

10 11 12

15 13 14

16 17 18

→ go to every row,
and print that row by travelling each column;

rows = 3
cols = 3

```
for (let r = 0; r < rows; r++) {
    for (let c = 0; c < cols; c++) {
        psw (m[r][c] + " ");
    }
    console.log();
}
```

① r = 0
→ c = 0 → m(0)(0)
→ c = 1 → m(0)(1)
→ c = 2 → m(0)(2)

② r = 1
→ c = 0 → m(1)(0)
→ c = 1 → m(1)(1)
→ c = 2 → m(1)(2)

③ r = 2
→ c = 0
→ c = 1
→ c = 2

④ r = 3 ✗
(3 < 3)

Q: Create a N x M matrix in the following pattern !

eg: Ip, n = 3, m = 4

op:

	0	1	2	3
0	1	2	3	4
1	1	2	3	4
2	1	2	3	4

3x4

```
matrix = [];
for (let r=0; r < rows; r++) {
    const smallArr = [];
    for (let c = 0; c < cols; c++) {
        smallArr.push(c+1);
    }
    matrix.push(smallArr);
}
```

0 1 2

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

→ To Build an entire matrix,
= first you need to build smaller arrays

4 cols

3 rows

const arr1 = [1,2,3,4]
const arr2 = [1,2,3,4]
const arr3 = [1,2,3,4]

① Build small

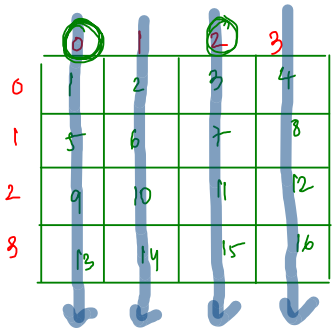
const mat = [arr1, arr2, arr3]

② store smaller in bigger

→ c=0, [1]
→ c=1, [1,2]
→ c=2, [1,2,3]
→ c=3, [1,2,3,4]
→ c=4 (4<4) X

r=0 [1,2,3,4]
r=1 [1,2,3,4]
[1,2,3,4]

★ Print matrix column wise :



op: 1 5 9 13 2 6 10 14
3 7 11 15 4 8 12 16

★ fix col
⟶ go to every row

r=0, c=0 r=0, c=1
r=1, c=0 r=1, c=1
r=2, c=0 r=2, c=1 c=2
r=3, c=0 r=3, c=1 r=0→3

c=0 c=1 c=3
r=0→3 r=0→3 r=0→3

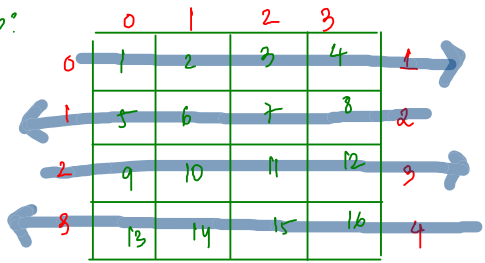
```
for(let c=0; c<cols; c++) {
    for(let r=0; r<rows; r++) {
        psw(m[r][c] + " ");
    }
}
```

O(rows) O(N) for every col ⇒ O(N×M)

★ TC : O(rows × cols) ⇒ O(N×M)
SC : O(1)

★ Alternate Matrix traversal :

ip:



odd Rows
(1st, 3rd, 5th, etc ...)
(L → R)

even Rows
(2nd, 4th, 6th, etc ....)
(R → L)

⇕

In 0-indexing terms

odd Rows
(1st, 3rd, 5th, etc ...)
(R → L)

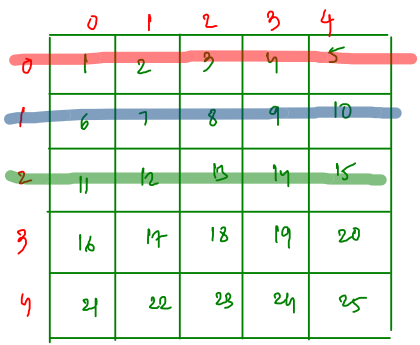
even Rows
(0th, 2nd, 4th, etc ...)
(L → R)

★ TC : O(n×m)

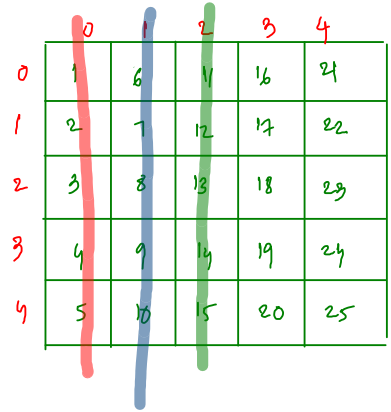
★ SC : O(1)

```
for (let r = 0; r < rows; r++) {
    if (r % 2 == 0) {
        for (let c = 0; c < cols; c++) {
            psw(m[r][c] + " ");
        }
    }
    else {
        for (let c = cols-1; c >= 0; c--) {
            psw(m[r][c] + " ");
        }
    }
}
```

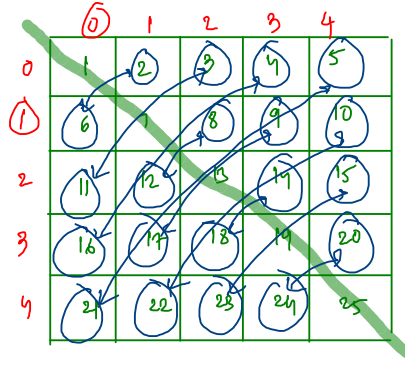
★ Transpose of a Matrix : [In-place]



Interchange
rows ⟷ cols



$m(r)(c) \longleftrightarrow m(c)(r)$



$m(1)(0) \longleftrightarrow m(0)(1)$

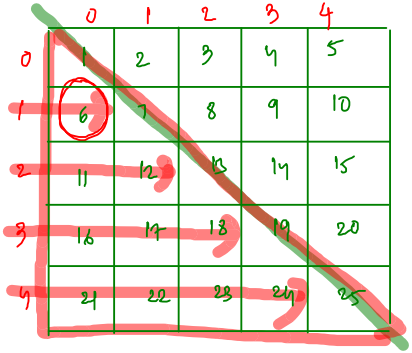
$m(2)(0) \longleftrightarrow m(0)(2)$

$m(2)(1) \longleftrightarrow m(1)(2)$

$m(3)(0) \longleftrightarrow m(0)(3)$

$m(3)(1) \longleftrightarrow m(1)(3)$

$m(3)(2) \longleftrightarrow m(2)(3)$



r = 0 → 1 time
r = 1 → 2 times
r = 2 → 3 times
r = 3 → 4 times
r = 4 → 5 times

rows ⇒ 1 + 2 + 3 + 4 + 5
n rows ⇒ 1 + 2 + 3 + . . . + n
⇒ $\frac{n(n+1)}{2}$ ⇒ O(N²), SC: O(1)

r = 0, c = 0
r = 1, c = 0, 1
r = 2, c = 0, 1, 2
r = 3, c = 0, 1, 2, 3
r = 4, c = 0, 1, 2, 3, 4

} c = 0 ⟶ r

```
for (let r = 0; r < rows; r++) {
    for (let c = 0; c <= r; c++) {
        let temp = m(r)(c);
        m(r)(c) = m(c)(r);
        m(c)(r) = temp;
    }
}
```

* N × M matrix will not work as this doesn't have a diagonal, hence we will use the extra space approach

```
function hw1(n) {
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < i; j++) {
      console.log("*");
      break;
    }
  }
}

function hw2(n) {
  let i = 1;
  while (i ** 2 <= n) {
    i = i + 1;
  }
}

function hw3(m, n) {
  while (m != n) {
    if (m > n) {
      m = m - n;
    } else {
      n = n - m;
    }
  }
}
```

```
function hw4(n) {
  let i = 1;
  while (i < n) {
    let j = n;
    while (j > 0) {
      j = parseInt(j / 2);
    }
    i = i * 2;
  }
}

function hw5(n) {
  for (let i = 0; i < parseInt(n / 2); i++) {
    for (let j = 1; j < n - parseInt(n / 2); j++) {
      let m = 1;
      while (m <= n) {
        m = m * 2;
      }
    }
  }
}
```