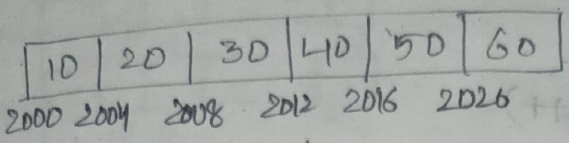
Multidimensional Arrays / Matrices

Important points:

① elements are stored in row-major order.



② Internal curly braces are optional.

int arr[3][2] = {10, 20, 30, 40, 50, 60};

③ only the first dimension can be omitted when we initialize

int arr[][2] = [[1,2],[3,4]]

int arr[][2][2] = [[[1,2],[3,4]],
[[5,6],[7,8]]]

```
#include <iostream>
using namespace std;
int main()
{
int arr[3][2] = {{10,20},
                 {30,40},
                 {50,60}};
for (int i=0; i<3; i++)
    for (int j=0; j<2; j++)
        cout << arr[i][j] << " ";
return 0;
}
```

o/p: 10 20 30 40 50 60

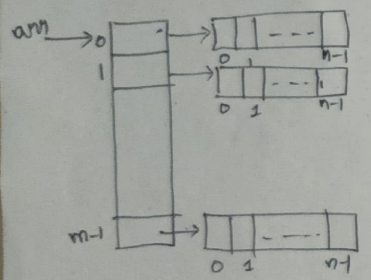
Variable sized:

```
int main()
{
  int m=3, n=2;
  int arr[m][n];
  for(int i=0; i<m; i++)
    for (int j=0; j<n; j++)
      arr[i][j] = i+j;
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
      cout << arr[i][j] << " ";
}
```

o/p: 0 1 1 2 2 3

other ways to create:

① Double pointer:



(Heap)

```
int main()
{
  int m=3, n=2;
  int **arr;
  arr = new int *[m];
  for (int i=0; i<m; i++)
    arr[i] = new int [n];
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
    { arr[i][j] = 10;
      cout << arr[j][j] << " ";
    }
}
```

m x n

dynamic allocated

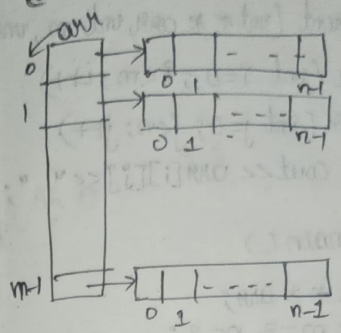
o/p: 10 10 10 10 10 10

advantages:
① create an jagged array of individual size passing in place of n.

disadvantages:
① not cache friendly

② Array of pointers

(allocated on stack)




```
int main()
{
  int m=3, n=2;
  int *arr[m];
  for (int i=0; i<m; i++)
    arr[i] = new int[n];
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
    { arr[i][j]=10;
      cout << arr[i][j] << " ";
    }
}
```

m x n

③ Array of vectors:

① Not as cache friendly as simple 2-D arrays.
② Individual rows are of dynamic sizes.
③ easy to pass to a function

```
int main()
{
  int m=3, n=2;
  vector<int> arr[m];
  for(int i=0; i<m; i++)
    for(int j=0; j<n; j++)
      arr[i].push_back(10);
  for(int i=0; i<m; i++)
    for (int j=0; j<n; j++)
      cout << arr[i][j] << " ";
}
```

o/p: 10 10 10 10 10 10

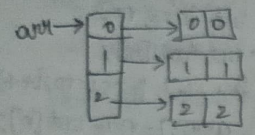
④ vector of vectors

Number of rows can also be dynamic

```
int main()
{ int m=3, n=2;
  vector<vector<int>> arr;
  for(int i=0; i<m; i++)
  { vector<int> v;
    for(int j=0; j<n; j++)
      v.push_back(10);
    arr.push_back(v);
  }
  for(int i=0; i<m; i++)
    for(int j=0; j<n; j++)
      cout << arr[i][j] << " ";
}
```

C-style solutions to create general purpose matrix method:

① using double pointer:-




```
void print (int** arr, int m, int n)
{ for (int i=0 ; i<m ; i++)
    for (int j=0; j<n; j++)
      cout<< arr[i][j]<<" ";
}
int main ()
{
  int** arr;
  int m=3, n=2;
  arr = new int*[m];
  for(int i=0; i<m;i++)
  { arr[i]= new int [n];
    for (int j=0; j<n; j++)
    { arr[i][j] = i;
      cout << arr[i][j]<<" ";
    }
  return 0;
}
```

o/p: 0 0 1 1 2 2

② using array of pointers:-

```
void print (int *arr[], int m, int n)
{ for (int i=0; i<m ; i++)
    for (int j=0; j<n; j++)
      cout << arr[i][j]<<" ";
}
int main ()
{ int m=3, n=2;
  int *arr[m];   → Array of pointers
  for (int i=0; i<m; i++)
  { arr[i]=new int[n];
    for (int j=0; j<n;j++)
    { arr[i][j]=1;
      cout<< arr[i][j]<<" ";
}}}
```

o/p: 0 0 1 1 2 2

C++ styles

① using array of vectors:-

```
void print (vector<int>arr[], int m)
{ for (int i=0; i<m; i++)
    for (int j=0; j<arr[i].size(); j++)
      cout << arr[i][j] << " ";
}
int main ()
{
  int m=3, n=2;
  vector<int> arr[m];
  for (int i=0; i<m; i++)
    for (int j=0; j<n; j++)
      arr[i].push_back (i);
  print (arr);
  return 0;
}
```

o/p: 0 0 1 1 2 2

② using vector of vectors:-

Jagged Array

```
void print (vector<vector<int>>&arr)
{ for (int i=0; i<arr.size(); i++)
    for (int j=0; j<arr[i].size(); j++)
      cout << arr[i][j] << " ";
}
int main()
{ int m=3, n=2;
  vector<vector<int>> arr;
  for (int i=0; i<m; i++)
  { vector<int> v;
    for (int j=0; j<n; j++)
      v.push_back(i);
    arr.push_back(v);
  }
  print(arr);
}
```

o/p: 0 0 1 1 2 2

Print a matrix in snake pattern:

I/P:

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

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

I/P:

1 2 3 4
5 6 7 8
9 10 11 12

O/P: 1 2 3 4 8 7 6 5 9 10 11 12

Pseudo code:

$T.C = \theta(R * C)$

```
void printSnake(int mat[R][C])
{
  for (int i=0; i<R; i++)
  {
    if (i%2==0)
    {
      for (int j=0; j<C; j++)
        print (mat[i][j] + " ");
    }
    else
    {
      for (int j=C-1; j>=0; j--)
        print (mat[i][j] + " ");
    }
```

Rows: 0, 2, ---

Rows: 1, 3, ---

Print Boundary Elements:

I/P:

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

O/P: 1 2 3 4 8 12 16 15 14 13 9 5

I/P:

1 2 3 4
5 6 7 8

O/P: 1 2 3 4 8 7 6 5

I/P: 1 2 3 4

O/P: 1 2 3 4

I/P:

1
2
3

O/P: 1 2 3

Corner cases:

1 2 3 4

1
2
3
4

```
Void bTraversal (int mat[R][C])
{
  if (R==1)
  {
    for (int i=0; i<C; i++)
    {
      print (mat[0][i] + " ");
    }
  }
  else if (C==1)
  {
    for (int i=0; i<R; i++)
      print (mat[i][0] + " ");
  }
  else
  {
    for (int i=0; i<C; i++)
      print (mat[0][i] + " ");
    for (int i=1; i<R; i++)
      print (mat[i][C-1] + " ");
    for (int i=C-2; i>=0; i--)
      print (mat[R-1][i] + " ");
    for (int i=R-2; i>=1; i--)
      print (mat[i][0] + " ");
  }
}
```

I/P:

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

O/P: 1 2 3 4 8 12 16 15 14 13 9 5

T.C = ~~$\theta(R)+\theta(R)+\theta(C)+\theta(C)$~~
= $\theta(R+R+C+C)$
= $\theta(2R+2C)$
T.C = $\theta(R+C)$

Transpose of a matrix:

(n x n)

I/P:

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

O/P:

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

I/P:

1	1
2	2

O/P:

1	2
1	2

Naive solution:

```
void transpose (int mat [n][n])
{
  int temp[n][n];
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      temp[i][j] = mat[j][i]; // temp[j][i]=mat[i][j]
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      mat[i][j] = temp[i][j];
}
```

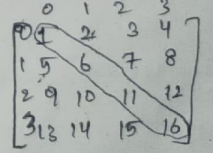
T.C = $\theta(n^2)$
A.S = $\theta(n^2)$

Effecient solution: Inplace -> one traversal

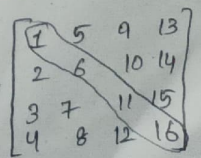
```
void transpose (int mat[n][n])
{
  for (int i=0; i<n; i++)
    for (int j=i+1; j<n; j++)
      swap(mat[i][j], mat[j][i]); // transpose
}
```

T.C = O(n*n)

I/P:



O/P:



Rotate a matrix by 90° Anticlockwise

I/p:
1 2 3
4 5 6
7 8 9

o/p:
3 6 9
2 5 8
1 4 7

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

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

Naive solution: T.C = $O(n^2)$, A.S = $O(n^2)$

→ Last column becomes first row.
→ Second last column becomes second row.

```
void rotate90(int mat[R][C])
{ int temp[n][n];
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      temp[n-j-1][i] = mat[i][j]
  for (int i=0; i<n; i++)
    for (int j=0; j<n; j++)
      mat[i][j] = temp[i][j];
}
```

Efficient solution: T.C = $O(n^2)$, A.S = $O(1)$

① find transpose of matrix
② Reverse individual columns

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

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

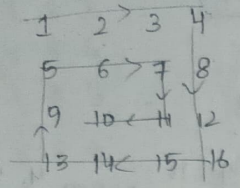
Reverse →
4 8 12 16
3 7 11 15
2 6 10 14
1 5 9 13

```
void rotate90(int mat[n][n])
{ for (int i=0; i<n; i++)
    for (int j=i+1; j<n; j++)
      Swap(mat[i][j], mat[j][i]);        // Transpose

  for (int i=0; i<n; i++)
  { int low=0, high=n-1;
    while (low<high)
    { swap(mat[low][i], mat[high][i]);   // Reverse columns
      low++;
      high--;
    }
  }
}
```

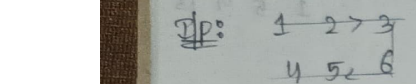
Spiral Traversal of a matrix

I/p:



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

I/p:



O/p: 1 2 3 6 5 4

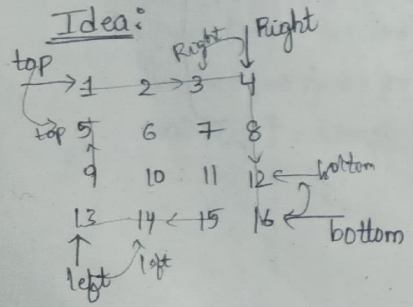
I/p: 10 20 30

O/p: 10 20 30

I/p: 10
20
30

O/p: 10 20 30

Idea:



Top row
Right column
Bottom row (reverse)
Left column (reverse)

Pseudo code:

```
void printSpiral (int mat[][], int R, int C)
{ int top=0, left=0, bottom=R-1, Right=C-1;
  while (top<=bottom && left <= Right)
  {
    for (int i=left; i<=Right; i++)
        print (mat[top][i] + " ");
    top++;                                   } Top Row

    for (int i=top; i<=bottom; i++)
        print (mat[i][Right] + " ");         } Right column;
    Right --;

    if (top<=bottom){
      for (int i=right; i>=left; i--)
          print (mat[bottom][i]+" ");        } Bottom Row (reverse order)
      bottom--;
    }
    if (left<=right)
    { for (int i=bottom; i>=top; i--)
          print (mat[i][left] + " ");        } Left column (reverse order)
      left++;
    }
```

T.C = O(R * C)

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

Initially: top=0 left=0 bottom=3 right=3

o/p: 1 2 3 4, top=1

o/p: 1 2 3 4 8 12 16, ~~bottom~~ right=2

o/p: 1 2 3 4 8 12 16 15 14 13, bottom=2

o/p: 1 2 3 4 8 12 16 15 14 13 9 5, left=1

o/p: 1 2 3 4 8 12 16 15 14 13 9 5 6 7, top=2

o/p: 1 2 3 4 8 12 16 15 14 13 9 5 6 7 11, right=1

o/p: 1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10, bottom=1

Search in Row wise and column wise Sorted matrix

I/p: mat[][] = { {10, 20, 30, 40},
{15, 25, 35, 45},
{27, 29, 37, 48},
{32, 33, 39, 50} }

x = 29

o/p: Found at (2,1)

I/p: mat[][] = { {10, 20},
{15, 30} }

x = 15

o/p: Not Found.

Naive : O(RXC) :

```
void search (int mat[R][C], int x)
{ for (int i=0; i<R; i++)
  { for(int j=0; j<C; j++)
    { if (mat[i][j] == x)
      { print("Found at ("+i+","+j+")");
        return;
      }
    }
  }
  print ("not Found");
}
```

Efficient Solution : TC: O(R+C) :

optimization: 1) if element searched is < top left return not found. 2) if search is > bottom right return not found

→ Begin from the top right corner.
→ if x is same, print position and return.
→ if x is smaller, move left
→ if x is greater, move ~~right~~ down.

Why top right?

```
void search( int mat[R][C], int x)
{ int i=0, j=C-1;
  while (i<R && j>=0)
  { if(mat[i][j] == x)
    { print ("Found at ("+i+","+j+")");
      return;
    }
    else if (mat[i][j] > x)
      j--;
    else
      i++;
  }
  print ("Not Found");
}
```

mat[][] =
10	20	30	40
15	25	35	45
27	29	37	48
32	33	39	50

x = 29.

to dry run: x = 24

Initially: i=0, j=3
After Iteration ① : j=2
After Iteration ② : j=1
After Iteration ③ : i=1
After Iteration ④ : i=2
In Iteration ⑤ : found at (2,2)

Median of a Row-wise Sorted Matrix

Assumptions:

* odd sized matrix
* All distinct elements

I/p: mat[][] = $\begin{bmatrix} 1 & 10 & 20 \\ 15 & 25 & 35 \\ 5 & 30 & 40 \end{bmatrix}$

O/p: 20 // 1, 5, 10, 15, 20, 25, 30, 35, 40

I/p: mat[][] = $\begin{bmatrix} 2 & 4 & 6 & 8 & 10 \\ 1 & 3 & 5 & 7 & 9 \\ 100 & 200 & 400 & 500 & 800 \end{bmatrix}$

O/p: 8 // 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 100, 200, 400, 500, 800

Naive Solution: ⇒ T.C = O(r*c log(r*c)) | A.S = O(r*c)

① put all elements in an array

$\begin{bmatrix} 1 & 10 & 20 \\ 15 & 25 & 35 \\ 5 & 30 & 40 \end{bmatrix}$ → O(r*c)

arr[] = [1, 10, 20, 15, 25, 35, 5, 30, 40]

② Sort the array → O(r*c log(r*c))

arr[] = [1, 5, 8, 10, 15, 20, 25, 30, 40]

③ Return the middle element of the sorted array → O(1)

Efficient: O(r * log(max-min) * log c)

(max: maximum element, min: minimum element)

```
int matMed(int mat[][MAX], int r, int c)
{ int min = mat[0][0], max = mat[0][c-1];
  for (int i=1; i<r; i++)
  { if (mat[i][0] < min) { min = mat[i][0]; }
    if (mat[i][c-1] > max){ max = mat[i][c-1]; }
  }                                              } O(r)
  int medpos = (r*c+1)/2;
  while (min < max)
  {
    int mid = (min + max)/2;
    int midpos=0;
    for(int i=0; i<r; i++)
      midpos += upper_bound(mat[i], mat[i]+c, mid) - mat[i];
    if(midpos < medpos)
      min = mid+1;
    else
      max = mid;
  }
  return min;
}
```

} O(log(max-min) * r * log c/2)

min=11, max=20
mid=15
midpos = 2+5+3 = 10

min=11, max=15
mid=13
midpos = 2+5+2 = 9

min=11, max=13
mid=12
midpos = 2+5+1 = 8

min=11, max=12
mid=11
midpos = 2+5+1 = 8

min=11, max=11

medpos = 8

$\begin{bmatrix} 5 & 10 & 20 & 30 & 40 \\ 1 & 2 & 3 & 4 & 6 \\ 11 & 13 & 15 & 17 & 19 \end{bmatrix}$

min=1, max=40
mid=20
midpos = 3+5+5 = 13

min=1, max=20
mid=10
midpos = 2+5+0 = 7