```c
int coin_collect(int a[][100], int n){
  int i,j, coins[100][100];

  coins[0][0] = a[0][0]; //initial cell

  for (i=1; i<n; i++) //first row
    coins[0][i] = coins[0][i-1] + a[0][i];

  for (i=1; i<n; i++) //first column
    coins[i][0] = coins[i-1][0] + a[i][0];

  for (i=1; i<n; i++) //filling up the rest of the array
    for (j=1; j<n; j++)
      coins[i][j] = max(coins[i-1][j], coins[i][j-1])
                    + a[i][j];

  return coins[n-1][n-1]; //value of last cell
}
```

```c
int max(int a, int b){
  if (a>b) return a;
  else return b;
}

int main(){
  int m[100][100],i,j,n;

  scanf("%d", &n);
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      scanf("%d", &m[i][j]);

  printf("%d\n", coin_collect(m,n));
  return 0;
}
```

Passing two dimensional arrays as parameters

Write a program that takes a two dimensional array of type double [5][6] and prints the sum of entries in each row.

```
void marginals(double mat[5][6]) {
    int i,j; double rowsum;
    for (i=0; i < 5; i=i+1) {
        rowsum = 0.0;
        for (j=0; j < 6; j = j+1) {
          rowsum = rowsum+mat[i][j];
        }
       printf("%f ", rowsum);
    }
}
```

Question?

But suppose we have read only the first 3 rows out of the 5 rows of mat. And we would like to find the marginal sum of the first 3 rows.

Answer:

That's easy, we can take an additional parameter **nrows** and run the loop for i=0..(nrows-1) instead of 0..5.

The slightly generalized program would be:

```
void marginals(double mat[5][6], int nrows) {
    int i,j; double rowsum;
    for (i=0; i < nrows; i=i+1) {
        rowsum = 0.0;
        for (j=0; j < 6; j = j+1) {
          rowsum = rowsum+mat[i][j];
        }
       printf("%f ", rowsum);
    }
}
```

In parameter double mat[5][6], C completely ignores the number of rows 5. It is only interested in the number of cols: 6.

We declared mat to be of type double [5][6].
Does this mean that nrows should be <= 5?
We are not checking for it!

Let's see more examples…

The following program is exactly identical to the previous one.

```
void marginals(double mat[ ][6], int nrows)
 {
   int i,j; int rowsum;
   for (i=0; i < nrows; i=i+1) {
       rowsum = 0.0;
       for (j=0; j < 6; j = j+1) {
         rowsum = rowsum+mat[i][j];
       }
      printf("%f ", rowsum);
    }
}
```

1. Why? because C does not care about the number of rows, only the number of number of cols.
2. And why is that? We'll have to understand 2-dim array addressing.

This means that the above program works with a k X 6 matrix where k could be passed for nrows.

Example…

Example calls for marginals

```
void marginals(double mat[ ][6], int nrows);
void main() {
     double mat[9][6];
    /* read the first 8 rows into mat */
     marginals(mat,8);
}
```

```
void marginals(double mat[ ][6], int nrows);
void main() {
    double mat[9][6];
   /* read 9 rows into mat */
     marginals(mat,10);
}
```

UNSAFE

The 10th row of mat[9][6] is not defined. So we may get a segmentation fault when marginals() processes the 10th row, i.e., i becomes 9.

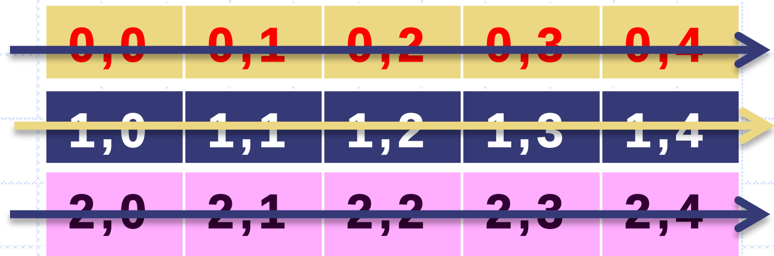
As with 1 dim arrays, allocate your array and stay within the limits allocated.

Why is # of columns required?

- The memory of a computer is a 1D array!
- 2D (or >2D) arrays are "flattened" into 1D to be stored in memory
- In C (and most other languages), arrays are flattened using Row-Major order
  - In case of 2D arrays, knowledge of number of columns is required to figure out where the next row starts.
  - Last n-1 dimensions required for nD arrays

Row Major Layout

mat[3][5]



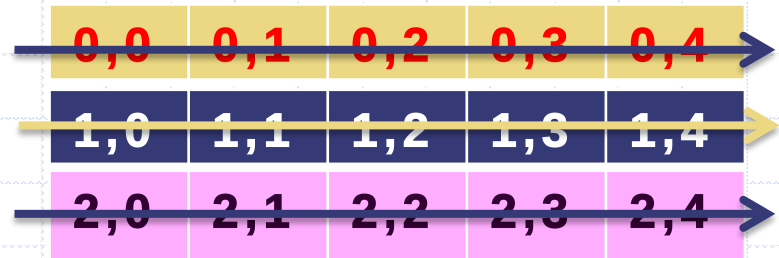
Layout of mat[3][5] in memory

- for a 2D array declared as **mat[M][N]**, cell [i][j] is stored in memory at location **i*N + j** from start of mat.
- for k-D array $arr[N_1][N_2]...[N_k]$, cell $[i_1][i_2]...[i_k]$ will be stored at location

$$i_k + N_k*(i_{k-1} + N_{k-1}*(i_{k-2} + (\ldots + N_2*i_1) \ldots))$$

Row Major Layout

mat[3][5]



Layout of mat[3][5] in memory

- **About C implementation:** a = *mat
- *mat = mat[0], *(mat+1) = mat[1], *(mat+2) = mat[2],...... Each of which stores the reference to the corresponding row.
- That is, **mat** POINTS to the beginning of the array that stores the references to each of the rows.

Array of Strings

- 2D array of char.
- Recall
  - Strings are character arrays that end with a '\0'
  - To display a string we can use printf with the %s placeholder.
  - To input a string we can use scanf with %s. Only reads non-whitespace characters.

Array of Strings: Example

- Write a program that reads and displays the name of few cities of India

```
const int ncity = 4;
const int lencity = 10;

int main(){
  char city[ncity][lencity];
  int i;

  for (i=0; i<ncity; i++){
    scanf("%s", city[i]);
  }

  for (i=0; i<ncity; i++){
    printf("%d %s\n", i, city[i]);
  }
  return 0;
}
```

INPUT
Delhi
Mumbai
Kolkata
Chennai

city[0]
city[1]

D	e	l	h	i	\0				
M	u	m	b	a	i	\0			
K	o	l	k	a	t	a	\0		
C	h	e	n	n	a	i	\0		

OUTPUT
0 Delhi
1 Mumbai
2 Kolkata
3 Chennai

Array of Strings: Example

- List initialization is also allowed:

```
const int ncity = 4;
const int lencity = 10;

int main(){
   char city[][lencity] = {“Delhi”,
    “Mumbai”, “Kolkata”, “Chennai”};
   int i;


   for (i=0; i<ncity; i++){
     printf("%d %s\n", i, city[i]);
   }
   return 0;
}
```

city[0]

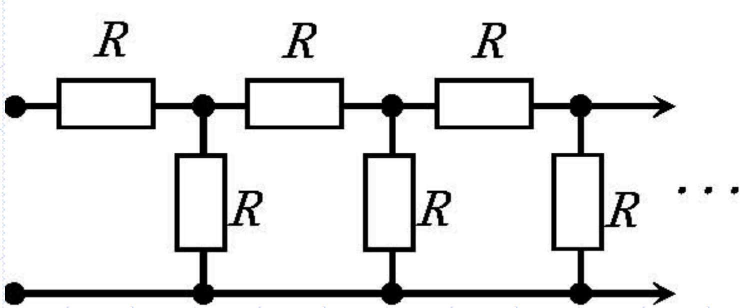
city[1]

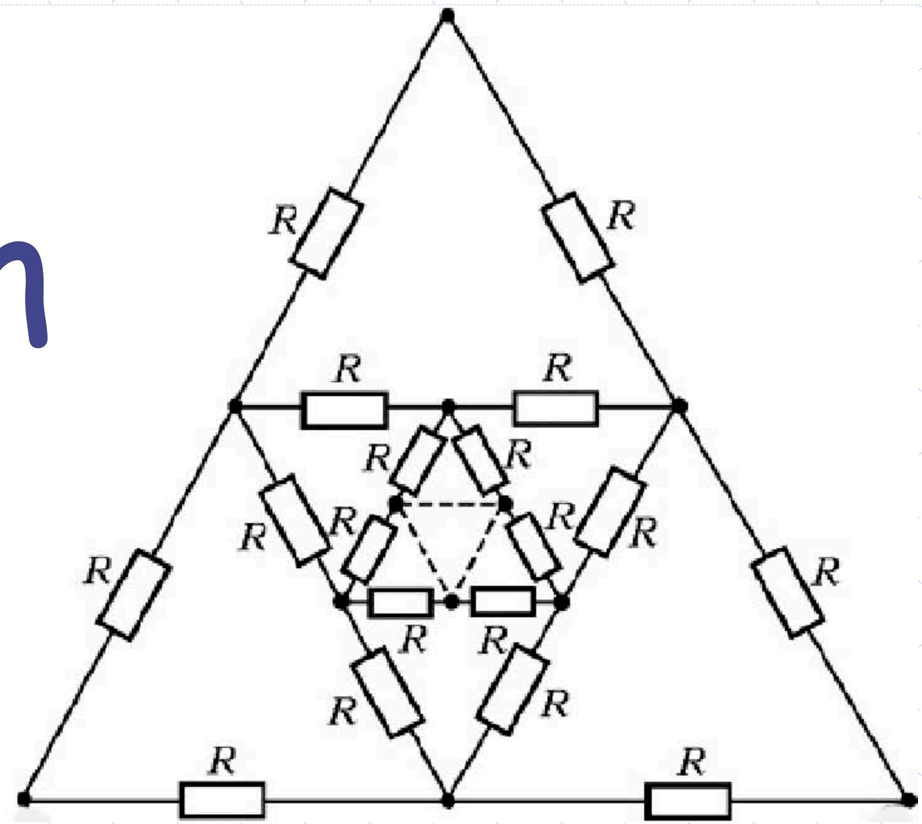
D	e	l	h	i	\0				
M	u	m	b	a	i	\0			
K	o	l	k	a	t	a	\0		
C	h	e	n	n	a	i	\0		

OUTPUT
0 Delhi
1 Mumbai
2 Kolkata
3 Chennai

ESC101: Introduction to Computing

Recursion





Recursion

- A function calling itself, *directly* or *indirectly*, is called a *recursive function.*
  - The phenomenon itself is called recursion
- Examples:
  - Factorial:

```
0! = 1
n! = n * (n-1)!
```

  - Even and Odd:

```
Even(n) = (n == 0) || Odd(n-1)
Odd(n)  = (n != 0) && Even(n-1)
```