

Unità didattica: Matrici come parametri dei sottoprogrammi [3-C]

Titolo: Matrici statiche e dinamiche nel passaggio dei parametri in C

Argomenti trattati:

- ✓ Le matrici statiche come parametri formali di function
- ✓ Le matrici dinamiche come parametri formali di function
- ✓ Efficienza degli algoritmi che accedono alle matrici assecondandone la corrispondente allocazione in memoria (allocazione per righe o per colonne)

Prerequisiti richiesti: programmazione C (matrici, puntatori, funzioni C per l'allocazione dinamica, function e parametri)

Le matrici (static array) come parametri formali

- Nel linguaggio **C** una matrice è memorizzata per righe e gli indici partono da 0.
- Il passaggio dei parametri è, in generale, *per valore* tranne quando un array è parametro ad una function: in tal caso viene passato l'**indirizzo base dell'array** cioè l'indirizzo della sua prima componente (in pratica il passaggio in tal caso è *per indirizzo*).
- Nei linguaggi **FORTRAN** e **MATLAB** una matrice è memorizzata per colonne e gli indici (per default) partono da 1.
- Il passaggio dei parametri è in generale *per indirizzo*.
- Come nel **C** quando una matrice è parametro di un sottoprogramma, viene passato l'**indirizzo base dell'array**.

In FORTRAN ...

dimensioni della **sottomatrice**
di A realmente usata

```
program matrici
integer A(4,5)
data A /1,4,0,0,2,5,0,0,3,6/
m=2; n=3;
print*,((A(i,j),j=1,n),i=1,m)
call matrice(m,n,A)
stop
end
```

```
C*****
subroutine matrice(mm, nn, M)
integer M(mm,nn)
print*,((M(i,j),j=1,nn),i=1,mm)
return
end
```

1	2	3	0	0
4	5	6	0	0
0	0	0	0	0
0	0	0	0	0

1	0	2
4	0	5

in memoria

1
4
0
0
2
5
0
0
0
0
0
0

RUN

1	2	3	4	5	6
1	0	2	4	0	5



In FORTRAN ...

dimensioni della **sottomatrice**
di A realmente usata

```
program matrici
integer A(4,5)
data A /1,4,0,0,2,5,0,0,3,6/
m=2; n=3; max_righe=4;
print*,((A(i,j),j=1,n),i=1,m)
call matrice(max_righe,m,n,A);
stop
end
```

1	2	3	0	0
4	5	6	0	0
0	0	0	0	0
0	0	0	0	0

LDA ?

1	2	3
4	5	6
0	0	0
0	0	0

in memoria

1	4	0	0	0	2	5	0	0	0	3	6	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Allocazione dinamica

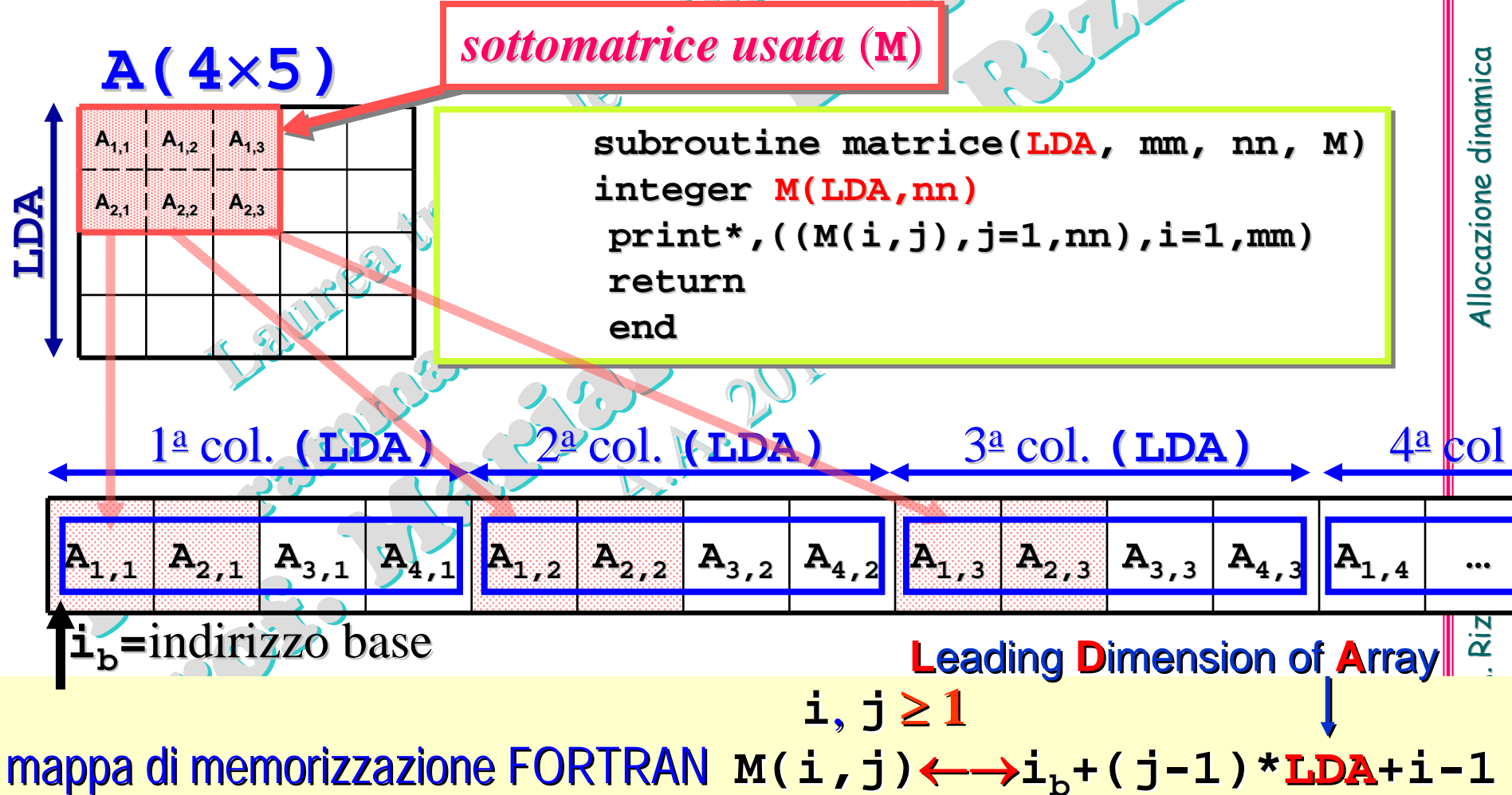
(prof. M. Rizzardi)

RUN

1	2	3	4	5	6
1	2	3	4	5	6



In **FORTRAN**, per una corrispondenza esatta tra gli elementi di **M** nella procedura e gli elementi di **A** nel programma principale è necessario dichiarare **LDA = numero di righe allocate** per la matrice nel programma principale anche se di questa se ne usa solo una sottomatrice.



In questo modo i sottoprogrammi **FORTRAN** possono essere scritti per matrici (statiche) di dimensioni fittizie, che solo al momento della chiamata si "sovrappongono idealmente" alle "vere" matrici (parametri attuali).

in **C**

in **C** le matrici sono memorizzate per righe



LDA = numero di colonne allocate

A(4x5)

A _{0,0}	A _{0,1}	A _{0,2}		
A _{1,0}	A _{1,1}	A _{1,2}		

LDA

1^a riga (**LDA**)

2^a riga (**LDA**)

3^a riga (**LDA**)

A _{0,0}	A _{0,1}	A _{0,2}	A _{0,3}	A _{0,4}	A _{1,0}	A _{1,1}	A _{1,2}	A _{1,3}	A _{1,4}	A _{2,0}	A _{2,1}	...
------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	-----

i_b = indirizzo base

Leading Dimension of Array

$i, j \geq 0$

mappa di memorizzazione **C** $M[i][j] \longleftrightarrow i_b + i * \text{LDA} + j$

```

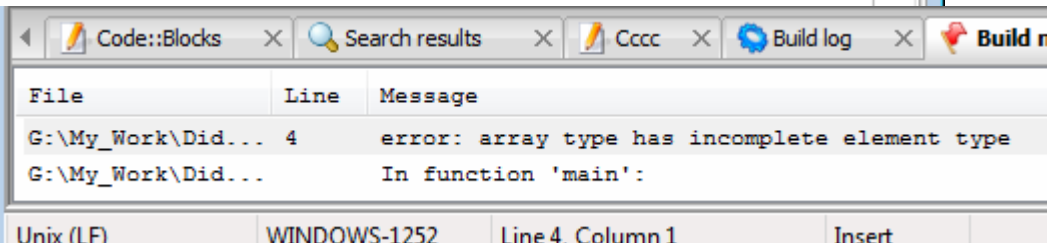
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void matrice(short lda, short mm, short nn, int M[][]);
5
6  main()
7  {
8  /** A = [1 2 3 _ _]
9          [4 5 6 _ _]
10         [_ _ _ _ _]
11         [_ _ _ _ _] */
12     int A[4][5] = {1,2,3,0,0,4,5,6};
13     short i,j, m=2, n=3, max_colonne=5; /* LDA */
14     for (i=0; i<m; i++)
15     {
16         for (j=0; j<n; j++)
17             printf("A[%d,%d]=%d,\t", i,j,A[i][j]);
18         printf("\n");
19     }
20     puts("\n");
21     matrice(max_colonne,m,n,A);
22     puts("\n");
23 }
24 /*****/
25 void matrice (short lda, short mm, short nn, int M[][lda])
26 {
27     short h,k;
28     for (h=0; h<mm; h++)
29     {
30         for (k=0; k<nn; k++)
31             printf("M[%d,%d]=%d,\t", h,k,M[h][k]);
32         printf("\n");
33     }
34 }

```

Errore

M	1	2	3	0	0
	4	5	6		

A



OK!

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void matrice(short lda, short mm, short nn, int M[][lda]);

```

```

5
6  main()
7  {
8      /** A = [1 2 3 _ _]
9              [4 5 6 _ _]
10             [_ _ _ _ _]
11             [_ _ _ _ _] */
12      int A[4][5] = {1,2,3,0,0,4,5,6};
13      short i,j, m=2, n=3, max_colonne=5; /* LDA */
14      for (i=0; i<m; i++)
15      {
16          for (j=0; j<n; j++)
17              printf("A[%d,%d]=%d,\t",i,j,A[i][j]);
18          printf("\n");
19      }
20      puts("\n");
21      matrice(max_colonne,m,n,A);
22      puts("\n");
23      /***/
24      void matrice(short lda, short mm, short nn, int M[][lda])
25      {
26          short h,k;
27          for (h=0; h<mm; h++)
28          {
29              for (k=0; k<nn; k++)
30                  printf("M[%d,%d]=%d,\t", h,k,M[h][k]);
31              printf("\n");
32          }
33      }
34

```

M	1	2	3	0	0
	4	5	6		

A

```

Code::Blocks  X  Search results  X  Cccc  X  Build log  X
Process terminated with status 0 (0 minute(s), 0 second(s))
0 error(s), 0 warning(s) (0 minute(s), 0 second(s))

```

```

Unix (LE)  WINDOWS 1252  Line 4, Column 1  Insert

```


FileEditViewSearchProjectBuildDebugFortranwxSmithToolsTools+PluginsDoxyBlocksSettingsHelp

parametroMatriceStatica1.c

1#include <stdio.h>

2#include <stdlib.h>

3

4void matrice(short lda, short mm, short nn, int M[][lda]);

5

6main()

7{

8/** A = [1 2 3 _ _]

9[4 5 6 _ _]

10[_ _ _ _ _]

11[_ _ _ _ _] */

12int A[4][5] = {1,2,3,0,0,4,5,6};

13short i,j, m=2, n=3, max_colonne=5; /*

14for (i=0; i<m; i++)

15{ for (j=0; j<n; j++)

16printf("A[%d,%d]=%d,\t",i,j,A[

17printf("\n");

18}

19puts("\n");

20matrice(max_colonne,m,n,A);

21puts("\n");

22}

23/*****

24void matrice(short lda, short mm, short nn, int M[][lda])

25{

26

27

28

29

30

31

Compiler settingsLinker settingsSearch directoriesToolchain executablesCustom variables

Policy:

Compiler FlagsOther options#defines

Categories:

<All categories>

☐ Profile code when executed [-pg]

☐ In C mode, support all ISO C90 programs. In C++ mode, remove GNU extensions that con

☒ Enable all common compiler warnings (overrides many other settings) [-Wall]

☐ Enable extra compiler warnings [-Wextra]

☐ Stop compiling after first error [-Wfatal-errors]

☐ Inhibit all warning messages [-w]

☐ Have g++ follow the 1998 ISO C++ language standard [-std=c++98]

☐ Have g++ follow the coming C++0x ISO C++ language standard [-std=c++0x]

☐ Have g++ follow the C++11 ISO C++ language standard [-std=c++11]

☐ Warn if '0' is used as a null pointer constant [-Wzero-as-null-pointer-constant]

☐ Enable warnings demanded by strict ISO C and ISO C++ [-pedantic]

☒ Treat as errors the warnings demanded by strict ISO C and ISO C++ [-pedantic-errors]

☐ Warn if main() is not conformant [-Wmain]

Code::BlocksSearch resultsCcccBuild logBuild messagesCppCheckC

File	Line	Message
G:\My_Work\Did...	4	error: ISO C90 forbids variable length array 'M' [-Wvla]
G:\My_Work\Did...	6	warning: return type defaults to 'int' [-Wreturn-type]

Unix (LF)WINDOWS-1252Line 4, Column 1InsertRead/Writedefault

compiler options

(prof. M. Rizzardi)

P2_05_03.9

il programma NON è standard (ISO C90)!

con le opzioni* di compilazione

* vedere: [options.htm](#) in Laboratorio

gcc -Wall -ansi -pedantic

```
...  
/*****  
void matrice(short lda, short mm, short nn, long M[][lda])  
{  
    short h, k;  
    for (h=0; h<mm; h++)  
        for (k=0; k<nn; k++)  
            printf("M[%d,%d]=%d,\t", h, k, M[h][k]);  
}
```

cosa richiede lo standard?

Il *linguaggio ISO C90* richiede che:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void matrice(short , short , short , int [][][5]);
5
6  int main()
7  {
8  /** A = [1 2 3 _ _]
9           [4 5 6 _ _]
10          [_ _ _ _ _]
11          [_ _ _ _ _] */
12  int A[4][5] = {{1,2,3,0,0},{4,5,6}};
13  short i,j, m=2, n=3, max_colonne=5; /* LDA */
14  for (i=0; i<m; i++)
15  {   for (j=0; j<n; j++)
16      printf("A[%d,%d]=%d,\t",i,j,A[i][j]);
17      printf("\n");
18  }
19  puts("\n");
20  matrice(max_colonne,m,n,A);
21  puts("\n");
22  return 0;
23 }
24
25 void matrice(short lda, short mm, short nn, int M[][5])
26 {
27     short h,k;
28     for (h=0; h<mm; h++)
29     {
30         for (k=0; k<nn; k++)
31             printf("M[%d,%d]=%d,\t",h,k,M[h][k]);
32         printf("\n");
33     }
34 }
```

costanti

inutile !!!

scomodo!

Come usare le matrici, tra i parametri delle function del *linguaggio C*, che possono essere usate da diversi programmi chiamanti per matrici di differenti dimensioni?

Bisogna usare i puntatori (!!!)

e preferibilmente ...
le matrici allocate dinamicamente.

...vedere: puntatori ad array 2D in Materiale utile!

Versione del programma con array statico

```
#include <stdio.h>
#include <stdlib.h>
```

```
void matrice(short , short , short , int *);
```

```
int main()
```

```
{/** A = [1 2 3 _ _]
          [4 5 6 _ _]
          [_ _ _ _ _]
          [_ _ _ _ _] **/
```

```
int A[4][5] = {{1,2,3,0,0},{4,5,6}}, (*pA)[4][5];
```

```
short i,j, m=2, n=3, LDA=5;
```

```
pA = &A;
```

```
for (i=0; i<m; i++)
```

```
{ for (j=0; j<n; j++)
```

```
    printf("\n");
```

```
}
```

```
puts("\n"); matrice(LDA,m,n, &(*pA)[0][0]); /*matrice(LDA,m,n, pA[0][0]); */
```

```
puts("\n");
```

```
return 0;
```

```
}
```

puntatore ad array 2D

(*pA)[4][5];

printf("(*pA)[%d][%d]=%d,\t",i,j,(*pA)[i][j]);

&(*pA)[0][0]); /*matrice(LDA,m,n, pA[0][0]); */

```
void matrice(short lda, short mm, short nn, int *pM)
```

```
{ short h,k;
```

```
for (h=0; h<mm; h++)
```

```
{ for (k=0; k<nn; k++)
```

```
    printf("(pM+%d*lda+%d)=%d,\t",h,k,*(pM+h*lda+k));
```

```
    printf("\n");
```

```
}
```

```
}
```

Versione del programma con array dinamico

più semplice!

```
#include <stdio.h>
#include <stdlib.h>
void matrice(short , short , int *);
```

```
int main()
```

```
{/** A = [1 2 3 _ _]
          [4 5 6 _ _]
          [_ _ _ _ _]
          [_ _ _ _ _] **/
```

```
short i,j, m=2, n=3;
```

```
int *pA = (int*)malloc(m*n*sizeof(int));
```

```
for (i=0; i<m; i++)
```

```
{ for (j=0; j<n; j++)
```

```
{ *(pA+i*n+j) = i*n+j+1;
```

```
printf("(pA+%d*n+%d]=%d,\t",i,j,*(pA+i*n+j));
```

```
}
```

```
printf("\n");
```

```
}
```

```
puts("\n"); matrice(m, n, pA);
```

```
puts("\n");
```

```
return 0;
```

```
}
```

```
void matrice(short mm, short nn, int *pM)
```

```
{ short h,k;
```

```
for (h=0; h<mm; h++)
```

```
{ for (k=0; k<nn; k++)
```

```
printf("(pM+%d*nn+%d)=%d,\t",h,k,*(pM+h*nn+k));
```

```
printf("\n");
```

```
}
```


Esercizio: differenza tra malloc() e calloc()

Ricordando che, date le matrici A e B di dimensioni rispettive $A(m \times p)$ e $B(p \times n)$ la matrice C , prodotto righe \times colonne $C = A \cdot B$, di dimensioni $(m \times n)$ è definita come

$$C_{ij} = \sum_{k=1}^p A_{ik} B_{kj}$$

scrivere una **function** C che restituisca la matrice C prodotto righe \times colonne delle due matrici rettangolari A e B le cui dimensioni sono stabilite in input (usare per tutte le matrici l'allocazione dinamica).

prodotto righe
per colonne


$$C = A \times B$$

help

[C'è qualche preferenza nell'usare malloc() o calloc() rispettivamente per A , B o C ?]

[I tempi d'esecuzione sono gli stessi?]

...vedere: `calcola_tempo.c` in Materiale utile!

Come calcolare il tempo di esecuzione di un blocco di istruzioni?

In `<time.h>`

funzioni per il tempo (poco accurate):

`time()`, `difftime()`
`clock()`, `CLOCKS_PER_SEC`

tutti i SO!

funzioni per il tempo (accurate ai nanosec):

`clock_gettime()`
`QueryPerformanceCounter()`

solo Linux

solo Windows

Esempio 1: confronto dei tempi di allocazione di calloc() e malloc()

```
#include <stdio.h>    #include <stdlib.h>    #include <time.h>
main()
{ int i,j, M=24000, N=20000; float *pA;
  time_t Time_start, Time_finish; double elapsed_Time;
  time( &Time_start );           /* tempo iniziale */
  pA = (float*)calloc(M*N,sizeof(float));
  time( &Time_finish );           /* tempo finale */
  if (pA == NULL)
  { fprintf(stderr, "\n*** Allocazione fallita! ***\n");
    exit(EXIT_FAILURE);
  }
  else
  { printf("\n*** Allocazione di float A(%d,%d): OK! ***\n", M,N);
    elapsed_Time = difftime(Time_finish, Time_start);
    printf("allocazione con calloc() richiede %g secondi\n", elapsed_Time);
    free(pA);
    time( &Time_start );           /* tempo iniziale */
    pA = (float*)malloc(M*N*sizeof(float));
    time( &Time_finish );           /* tempo finale */
    elapsed_Time = difftime(Time_finish, Time_start);
    printf("allocazione con malloc() richiede %g secondi\n", elapsed_Time);
    free(pA);
  }
}
```

```
*** Allocazione di float A(24000,20000): OK! ***
allocazione con calloc() richiede 0 secondi
allocazione con malloc() richiede 0 secondi
```

Esempio 2: confronto dei tempi di allocazione di calloc() e malloc()

```
#include <stdio.h>      #include <stdlib.h>      #include <time.h>
main()
{ int i, j, M=24000, N=20000; float *pA;
  clock_t ct1, ct2; double elapsed_Time;
  ct1 = clock();          /* tempo iniziale */
  pA = (float*)calloc(M*N, sizeof(float));
  ct2 = clock();          /* tempo finale */
  if (pA == NULL)
  { fprintf(stderr, "\n*** Allocazione fallita! ***\n");
    exit(EXIT_FAILURE);
  }
  else
  { printf("\n*** Allocazione di float A(%d,%d): OK! ***\n", M, N);
    elapsed_Time = (double)(ct2 - ct1)/(double)CLOCKS_PER_SEC;
    printf("allocazione con calloc() richiede %g secondi\n", elapsed_Time);
    free(pA);
    ct1 = clock();          /* tempo iniziale */
    pA = (float*)malloc(M*N*sizeof(float));
    ct2 = clock();          /* tempo finale */
    elapsed_Time = (double)(ct2 - ct1)/(double)CLOCKS_PER_SEC;
    printf("allocazione con malloc() richiede %g secondi\n", elapsed_Time);
    free(pA);
  }
}
```

```
*** Allocazione di float A(24000,20000): OK! ***
allocazione con calloc() richiede 0 secondi
allocazione con malloc() richiede 0 secondi
```

Win

Esempio 3: confronto dei tempi di allocazione di calloc() e malloc()

```
#include <stdio.h>    #include <stdlib.h>    #include <time.h>
#include <windows.h> ← non standard

main()
{ int i,j, M=24000, N=20000; float *pA;
  LARGE_INTEGER ticksPerSecond, TICKS1, TICKS2; double elapsed_Time;
  QueryPerformanceFrequency(&ticksPerSecond); // processor clock frequency
  QueryPerformanceCounter(&TICKS1); /* tempo iniziale */
  pA = (float*)calloc(M*N, sizeof(float));
  QueryPerformanceCounter(&TICKS2); /* tempo finale */
  if (pA == NULL)
  { fprintf(stderr, "\n*** Allocazione fallita! ***\n"); exit(EXIT_FAILURE);
  }
  else
    printf("\n*** Allocazione di float A(%d,%d): OK! ***\n", M,N);
  elapsed_Time = (double)(TICKS2.QuadPart -
                          TICKS1.QuadPart)/((double)ticksPerSecond.QuadPart;
  printf("allocazione con calloc() richiede %g secondi\n", elapsed_Time);
  free(pA);
  QueryPerformanceCounter(&TICKS1); /* tempo iniziale */
  pA = (float*)malloc(M*N*sizeof(float));
  QueryPerformanceCounter(&TICKS2); /* tempo finale */
  elapsed_Time = (double)(TICKS2.QuadPart-TICKS1.QuadPart)/((double)ticksPerSecond.QuadPart;
  printf("allocazione con malloc() richiede %g secondi\n", elapsed_Time);
  free(pA);
}
```

*** Allocazione di float A(24000,20000): OK! ***

allocazione con calloc() richiede 1.02336e-005 secondi

allocazione con malloc() richiede 7.67518e-006 secondi

Win

Esempio 4: confronto dei tempi di azzeramento di calloc() e malloc()

```
#include <stdio.h>    #include <stdlib.h>    #include <time.h>
main()
{
    int i,j, M=24000, N=20000; float *pA;
    time_t Time_start, Time_finish; double elapsed_Time;
    time( &Time_start );                /* tempo iniziale */
    pA = (float*)calloc(M*N,sizeof(float));
    time( &Time_finish );                /* tempo finale */
    if (pA == NULL)
    {
        fprintf(stderr, "\n*** Allocazione fallita! ***\n");
        exit(EXIT_FAILURE);
    }
    else
        printf("\n*** Allocazione di float A(%d,%d): OK! ***\n", M,N);
    elapsed_Time = difftime(Time_finish, Time_start);
    printf("azzeramento con calloc() richiede %g secondi\n", elapsed_Time);
    free(pA);
    time( &Time_start );                /* tempo iniziale */
    pA = (float*)malloc(M*N*sizeof(float));
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            *(pA+i*N+j) = 0.0f;
    time( &Time_finish );                /* tempo finale */
    elapsed_Time = difftime(Time_finish, Time_start);
    printf("azzeramento con malloc() richiede %g secondi\n", elapsed_Time);
    free(pA);
}
```

*** Allocazione di float A(24000,20000): OK! ***
azzeramento con calloc() richiede 0 secondi
azzeramento con malloc() richiede 1 secondi

Win

Esempio 5: confronto dei tempi di azzeramento di calloc() e malloc()

```
#include <stdio.h>    #include <stdlib.h>    #include <time.h>
main()
{
    int i, j, M=24000, N=20000; float *pA;
    clock_t ct1, ct2; double elapsed_Time;

    ct1 = clock(); /* tempo iniziale */
    pA = (float*)calloc(M*N, sizeof(float));
    ct2 = clock(); /* tempo finale */
    if (pA == NULL)
    {
        fprintf(stderr, "\n*** Allocazione fallita! ***\n");
        exit(EXIT_FAILURE);
    }
    else
        printf("\n*** Allocazione di float A(%d,%d): OK! ***\n", M, N);

    elapsed_Time = (double)(ct2 - ct1)/(double)CLOCKS_PER_SEC;
    printf("azzeramento con calloc() richiede %g secondi\n", elapsed_Time);

    free(pA);
    ct1 = clock(); /* tempo iniziale */
    pA = (float*)malloc(M*N*sizeof(float));
    for (i=0; i<M; i++)
        for (j=0; j<N; j++)
            *(pA+i*N+j) = 0.0f;
    ct2 = clock(); /* tempo finale */
    elapsed_Time = (double)(ct2 - ct1)/(double)CLOCKS_PER_SEC;
    printf("azzeramento con malloc() richiede %g secondi\n", elapsed_Time);
    free(pA);
}
```

```
*** Allocazione di float A(24000,20000): OK! ***
azzeramento con calloc() richiede 0 secondi
azzeramento con malloc() richiede 1.134 secondi
```

Win

Esempio 6: confronto dei tempi di azzeramento di calloc() e malloc()

```
#include <stdio.h>      #include <stdlib.h>      #include <time.h>
#include <windows.h> ← non standard

main()
{
    int i, j, M=24000, N=20000; float *pA;
    LARGE_INTEGER ticksPerSecond, TICKS1, TICKS2; double elapsed_Time;
    QueryPerformanceFrequency(&ticksPerSecond); // processor clock frequency
    QueryPerformanceCounter(&TICKS1); /* tempo iniziale */
    pA = (float*)calloc(M*N, sizeof(float));
    QueryPerformanceCounter(&TICKS2); /* tempo finale */
    if (pA == NULL)
    {
        fprintf(stderr, "\n*** Allocazione fallita! ***\n"); exit(EXIT_FAILURE);
    }
    else
    {
        printf("\n*** Allocazione di float A(%d,%d): OK! ***\n", M, N);
        elapsed_Time = (double)(TICKS2.QuadPart -
                                TICKS1.QuadPart) / ((double)ticksPerSecond.QuadPart);
        printf("allocazione con calloc() richiede %g secondi\n", elapsed_Time);
        free(pA);
        QueryPerformanceCounter(&TICKS1); /* tempo iniziale */
        pA = (float*)malloc(M*N*sizeof(float));
        for (i=0; i<M; i++)
            for (j=0; j<N; j++)
                *(pA+i*N+j) = 0.0f;
        QueryPerformanceCounter(&TICKS2); /* tempo finale */
        elapsed_Time = (double)(TICKS2.QuadPart -
                                TICKS1.QuadPart) / ((double)ticksPerSecond.QuadPart);
        printf("allocazione con malloc() richiede %g secondi\n", elapsed_Time);
        free(pA);
    }
}
```

*** Allocazione di float A(24000,20000): OK! ***
azzeramento con calloc() richiede 1.08732e-005 secondi
azzeramento con malloc() richiede 1.1458 secondi

Richiami: algoritmo del prodotto $r \times c$ di 2 matrici

```
for i:=1,2, ..., m
  for j:=1,2, ..., n
    C(i,j):=0.0
  endfor
endfor
```

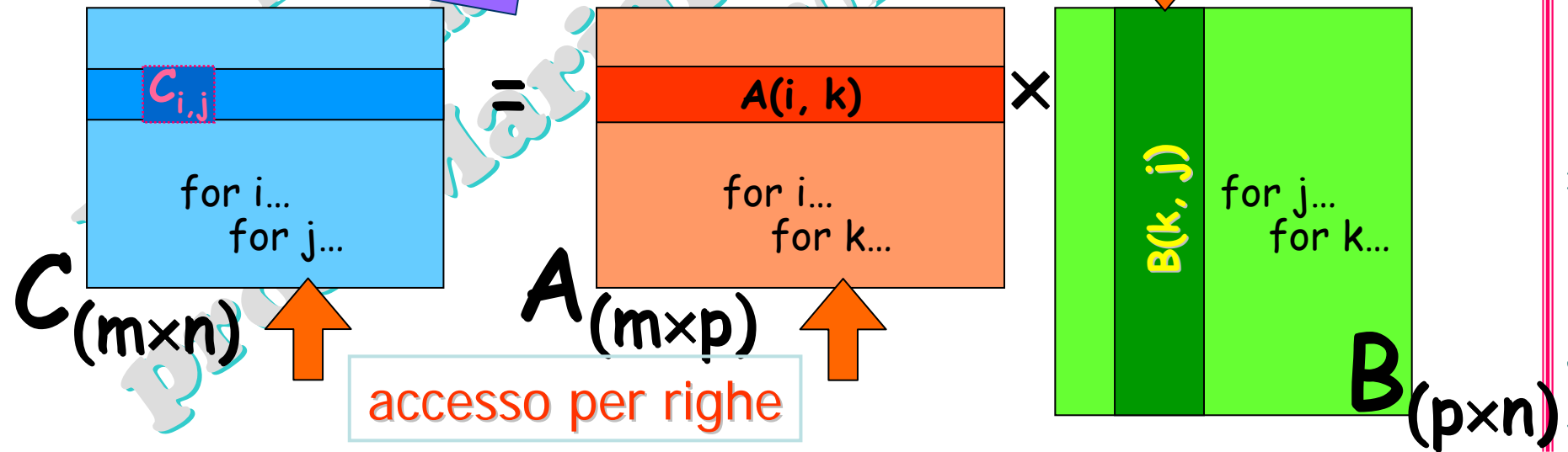
P-like

```
for i:=1,2, ..., m
  for j:=1,2, ..., n
    for k:=1,2, ..., p
      C(i,j):=C(i,j)+A(i,k)*B(k,j)
    endfor
  endfor
endfor
```

$$C_{(m \times n)} = A_{(m \times p)} \times B_{(p \times n)}$$

devono essere uguali

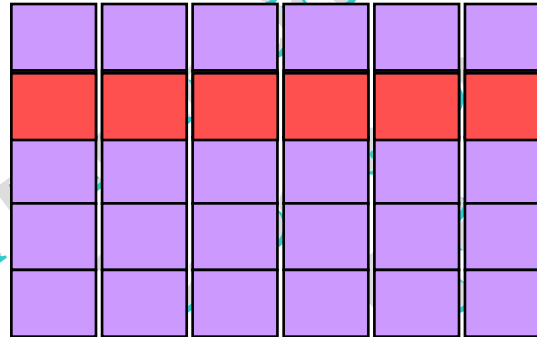
accesso per colonne



Accesso agli elementi di una matrice(*)

(*) indipendentemente da come è allocata in memoria

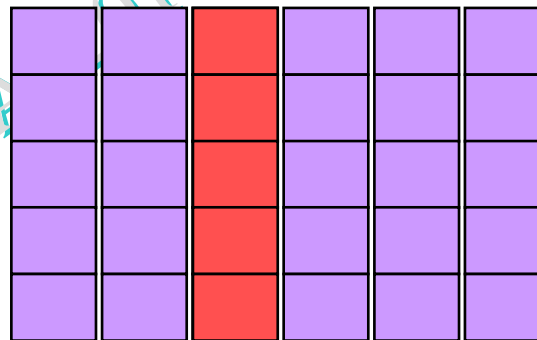
```
for i ...  
  for j ...  
    ...M(i,j)...  
  end  
end
```



$M_{(m \times n)}$

Accesso per righe alla matrice M :
quando j varia più velocemente di i in $M(i,j)$

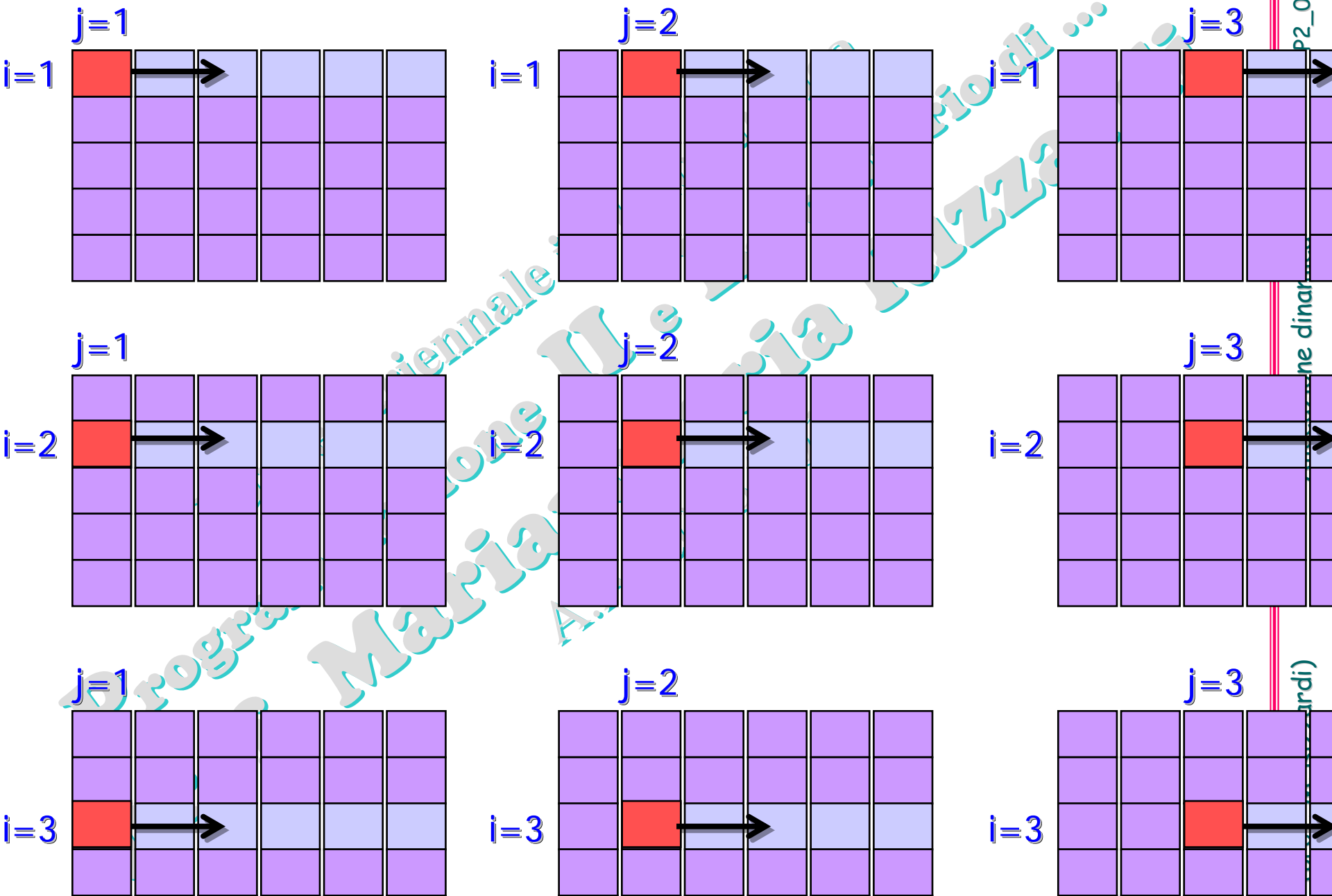
```
for j ...  
  for i ...  
    ...M(i,j)...  
  end  
end
```



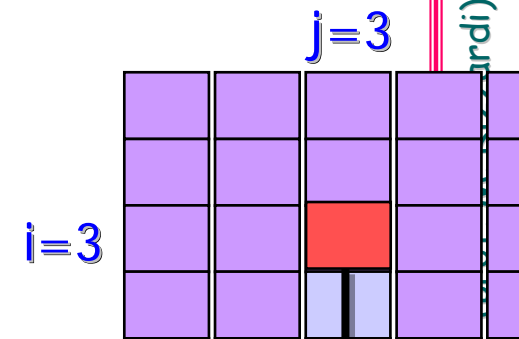
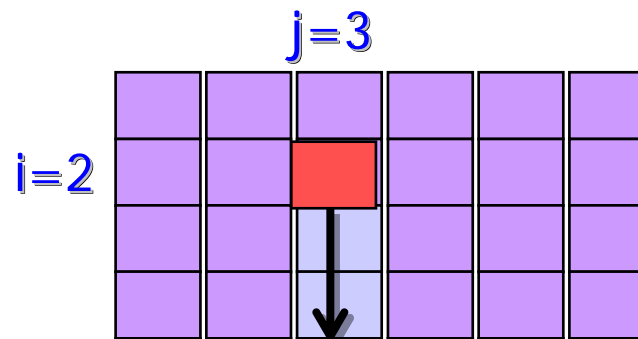
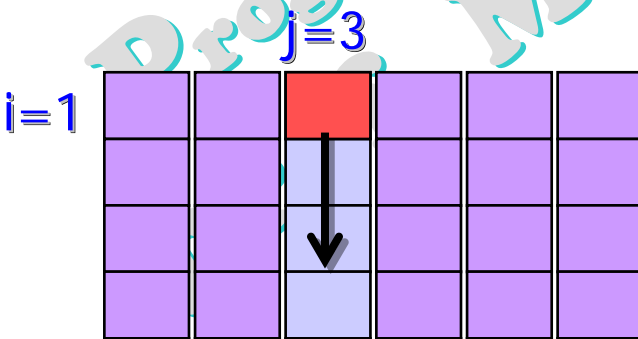
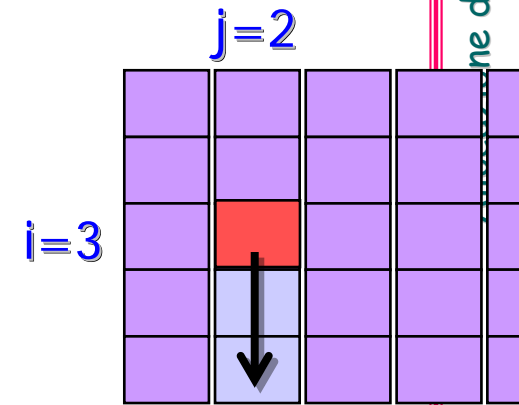
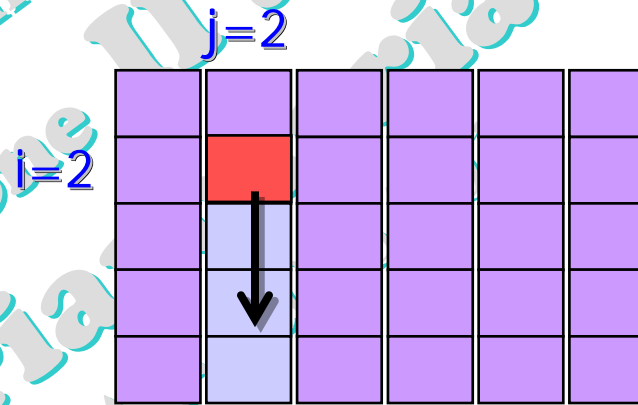
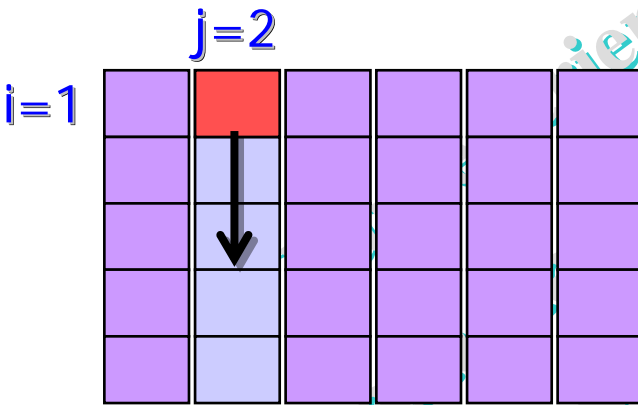
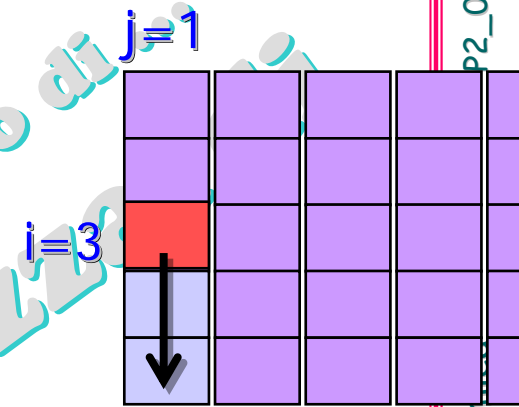
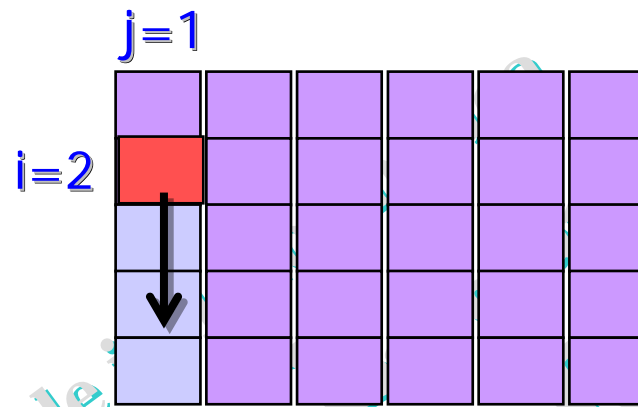
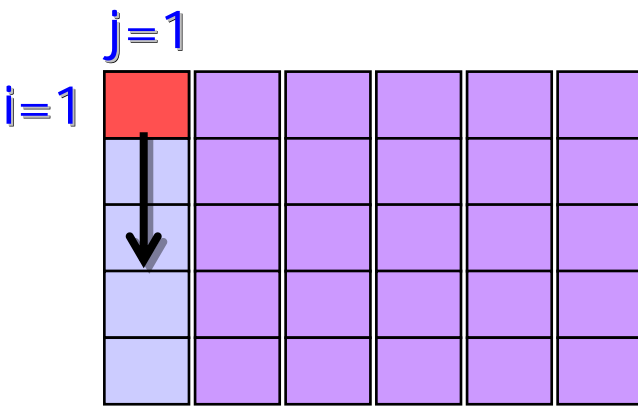
$M_{(m \times n)}$

Accesso per colonne alla matrice M :
quando i varia più velocemente di j in $M(i,j)$

Accesso per righe ad una matrice



Accesso per colonne ad una matrice



Come trasformare l'algoritmo affinché acceda a tutte le matrici solo per colonne oppure solo per righe(*)?

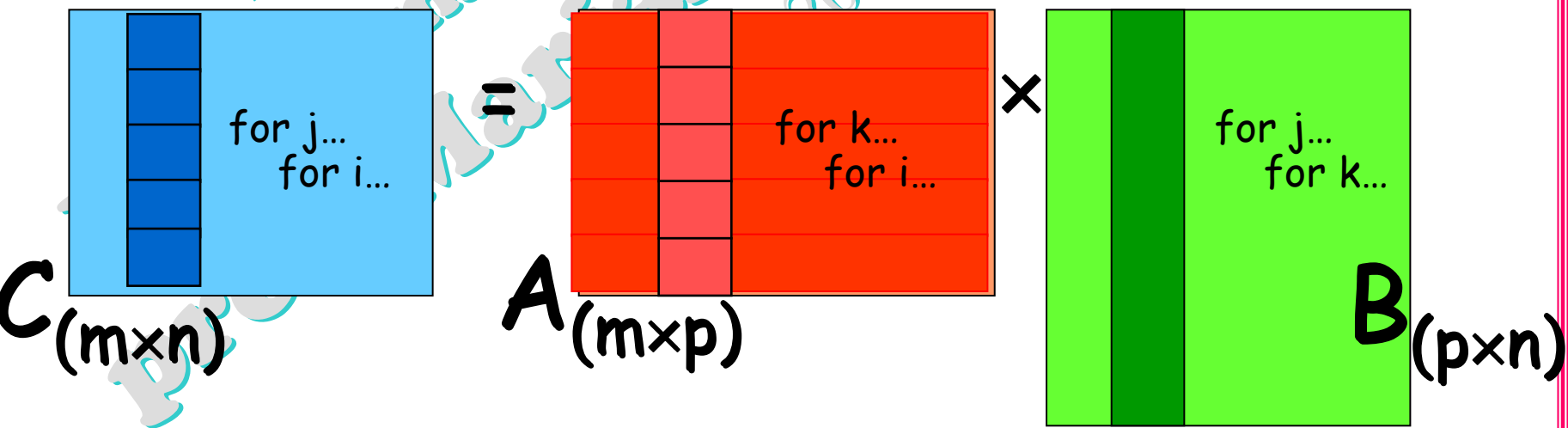
(*) indipendentemente da come sono allocate in memoria

```
for i ...
  for j ...
    ...M(i,j)...
  end
end
```

accesso per righe alla matrice **M**:
quando in **M(i,j)** **j** varia più velocemente di **i**

```
for j ...
  for i ...
    ...M(i,j)...
  end
end
```

accesso per colonne alla matrice **M**:
quando in **M(i,j)** **i** varia più velocemente di **j**



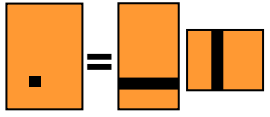
Esercizio: differenza tra

- **Allocazione** per righe o per colonne di una matrice.
- **Accesso** per righe o per colonne agli elementi di una matrice.

Ripetere l'esercizio sul prodotto righe x colonne allocando una prima volta tutte le matrici in memoria per colonne ed una seconda volta per righe. Per ciascun tipo di allocazione in memoria usare due *function* C per il prodotto righe x colonne: una che acceda a tutte le matrici per colonne e l'altra per righe.

Confrontare i **tempi d'esecuzione** delle due modalità di accesso alle matrici rispetto alla loro allocazione in memoria, deducendo quindi l'algoritmo e la memorizzazione più efficienti. **[liv.3]**

prodotto righe
per colonne


$$C = A \times B$$