# Architettura degli Elaboratori

## Funzioni, Ricorsione


SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Checco

checco@di.uniroma1.it

Special thanks and credits:

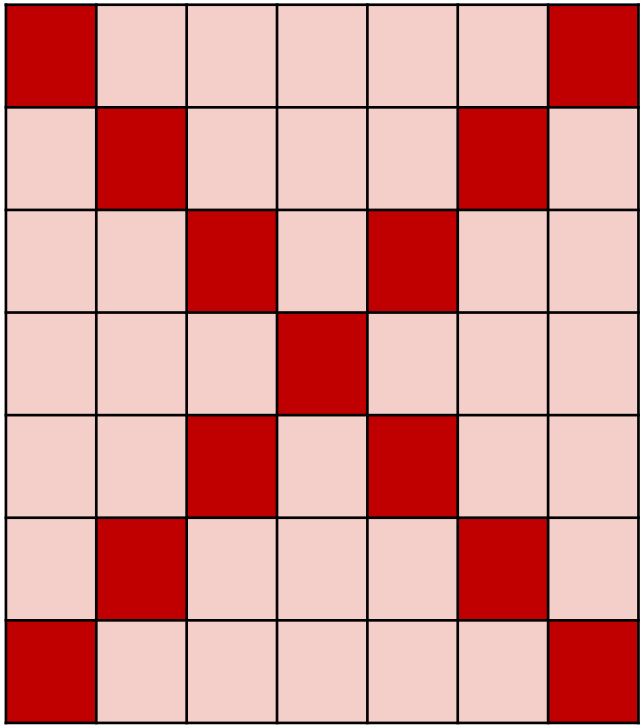
Andrea Sterbini, Iacopo Masi,
Claudio di Ciccio

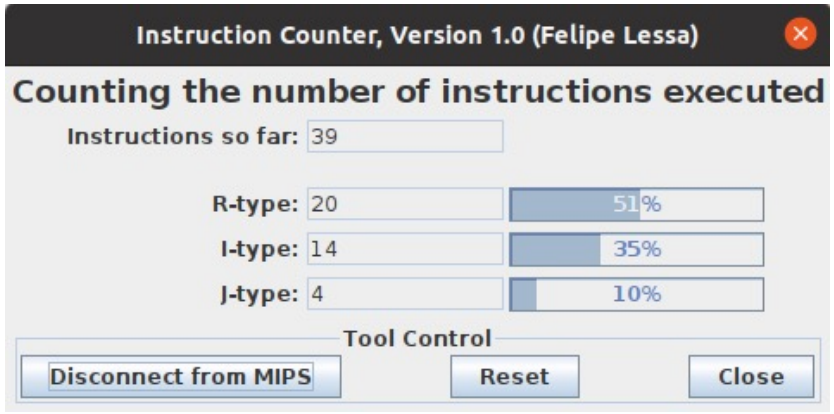
[S&PdC]
2.8–2.10, 2.14

# Esercizio


SAPIENZA
UNIVERSITÀ DI ROMA

# Somma della diagonale (v. più efficiente)



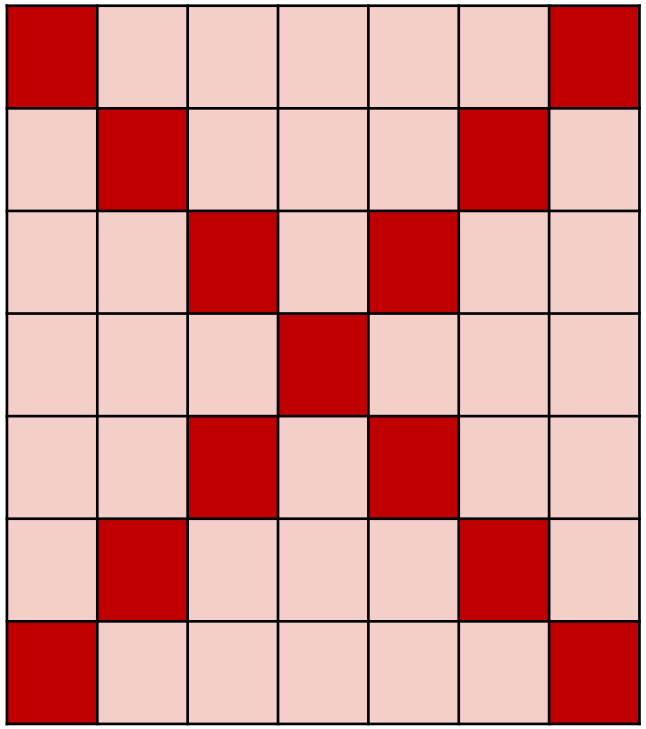


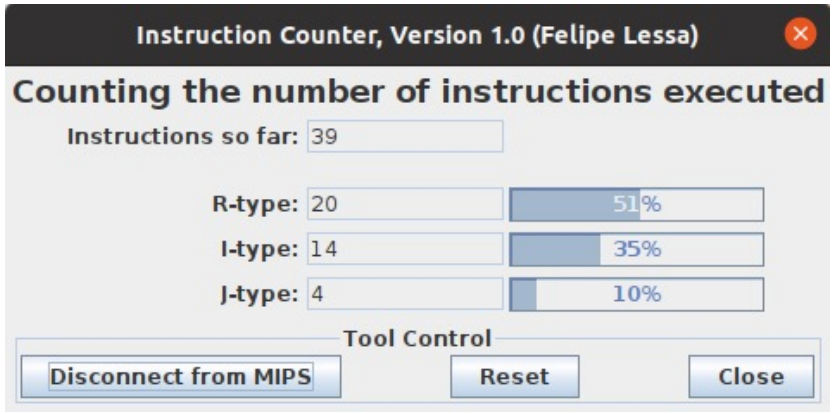

**Esercizio 1:**

Scrivere un programma in assembly che,

data una matrice bidimensionale quadrata MATRIX di word in memoria, di lato N, restituisca in `$s0` il risultato della somma dei valori sulle diagonali principale e secondaria.

# Somma della diagonale (v. più efficiente) / reprise






**Esercizio 2:**

Aggiungere istruzioni di stampa e di uscita al programma di cui all'esercizio 1.

Tutti I codici per syscall: https://courses.missouristate.edu/kenvollmar/mars/help/syscallhelp.html

# Soluzione es. somma diagonali con funzioni

Definiamo la funzione **is_diagonal** che torna **1** (vero) se la casella a coordinate **x, y** si trova su una delle due diagonali principali altrimenti torna **0** (falso), con argomenti:

```
#    $a0 coordinata x
#    $a1 coordinata y
#    $a2 lato della matrice
```

```
is_diagonal:
    beq    $a0, $a1, yes  # se x=y siamo sulla prima diagonale
    add    $v0, $a0, $a1  # altrimenti se la somma x+y+1
    addi   $v0, $v0, 1
    beq    $v0, $a2, yes  # è uguale al lato siamo sulla seconda
    # altrimenti non siamo sulle due diagonali principali
    li  $v0, 0  # il risultato è 0 (falso)
    jr  $ra     # ritorno all'istruzione successiva alla chiamata
yes:
    li  $v0, 1  # il risultato è 1 (vero)
    jr  $ra     # ritorno all'istruzione successiva alla chiamata
```

# lettura dell'elemento corrente

Definiamo una seconda funzione che legge dalla matrice l'elemento a coordinate **x,y**

(con gli stessi argomenti dell'altra)

```
#    $a0 coordinata x
#    $a1 coordinata y
#    $a2 lato della matrice
leggi_elemento:
   mul  $v0, $a1, $a2      # y*LATO
   add  $v0, $v0, $a0      # x + y*LATO
   sll  $v0, $v0, 2        # offset = 4 * (x + y * LATO)
   lw   $v0, matrice($v0)  # leggo matrice[x][y]
   jr   $ra                # torno  l'esecuzione al chiamante
```

**NOTA: cerco di non usare registri ulteriori in modo da lasciare più libertà nel programma chiamante**

**NOTA: stack vuoto, perché?**

# Programma principale

```
.data
   matrice: .word   0:400              # matrice 20x20
   LATO:    .word   20                 # lato della matrice
.text
main:       li      $a0, 0             # x = 0
            li      $a1, 0             # y = 0
            lw      $a2, LATO          # lato della matrice
            li      $t0, 0             # somma = 0
cicloY:     beq     $a1, $a2, fine     # se ultima riga
cicloX:     beq     $a0, $a2, nextY    # se ultima colonna
            jal     is_diagonal        # test se sulla diagonale
            beqz    $v0, nextX         # se falso prossima X
            jal     leggi_elemento     # altrimenti leggi
            add     $t0, $t0, $v0      # e somma l'elemento
nextX:      addi    $a0, $a0, 1        # x += 1 (prossima colonna)
            j       cicloX
```

# (segue)

```
nextY:      addi    $a1, $a1, 1     # y += 1 (prossima riga)
            li      $a0, 0          # x = 0  (colonna 0)
            j       cicloY
fine:       move    $a0, $t0        # stampo la somma
            li      $v0, 1          # syscall 1 = print integer
            syscall
            li      $v0, 10         # syscall 10 = fine
            syscall
```

# LA RICORSIONE


LA RICORSIONE
LA RICORSIONE
LA RICORSIONE
LA RICORSIONE
LA RICORSIONE

# Soluzioni ricorsive di problemi

**Quando si può usare una funzione ricorsiva per risolvere un problema?**

- se **esiste una soluzione conosciuta** per lo stesso problema di «piccole» dimensioni
    - da questa ricaviamo il **caso base** della funzione

- e se esiste un modo di ottenere la soluzione di un problema di dimensione maggiore **a partire dalla soluzione dello stesso problema di dimensione minore**
    - da questa definizione ricaviamo il **caso ricorsivo**, che è formato da 3 parti:
        - **riduzione** del problema in problemi «più piccoli»
        - **chiamata ricorsiva** della funzione per risolvere i casi «più piccoli»
        - elaborazione delle soluzioni «piccole» per ottenere la **soluzione del problema originale**

# Esempio

Funzione fattoriale

Definizione iterativa:

«Il fattoriale di un numero intero N è il prodotto dei numeri 1..N»

Ovvero:

```
risultato = 1
for (i=N; i>0; i--);
    risultato *= i
```

Definizione ricorsiva:

Caso base: «Il fattoriale di 1 è 1»

Riduzione del problema:

«Per moltiplicare i numeri da 1 a N posso prima moltiplicare da 1 a N-1…

Costruzione della soluzione finale a partire da quella ridotta:

… e poi moltiplicare per N»

ovvero

| | | |
|---|---|---|
| fattoriale(N) = 1 | se N < 2 | (caso base) |
| fattoriale(N) = N * fattoriale(N-1) | altrimenti | (caso ricorsivo) |

# Chiamate annidate

# Implementazione iterativa

```
factorial:
  subi $sp,$sp,4
  sw $a0,($sp)

  li $v0,1              # Initial result
  While:
    blez $a0, EndWhile # IF N <= 0 THEN exit the cycle
    mul $v0, $v0, $a0  # Multiply the result by N
    sub $a0, $a0, 1    # N ← N-1
    j While            # Loop back
  EndWhile:

  lw $a0,($sp)
  addi $sp,$sp,4
  jr  $ra              # Return
```

Esempio MARS

# Implementazione ricorsiva

```
factorial:
  blez $a0, BaseCase

  RecursiveStep:
    subi $sp,$sp,8        # Decrement the stack pointer
    sw $ra,0($sp)         # Store the return address for later
    sw $a0,4($sp)         # Store the value of n

    subi $a0,$a0,1        # Decrement n by 1
    jal factorial         # Do the recursive call

    lw $a0,4($sp)         # Restore n
    lw $ra,0($sp)         # Restore the return address
    addi $sp,$sp,8        # Restore the stack pointer

    mul $v0,$v0,$a0       # Compute n × factorial(n-1)

    jr $ra                # Return

  BaseCase:
    addi $v0,$zero,1      # 0! = 1! = 1

    jr $ra
```

# Implementazione ricorsiva (completa)

```
factorial:
  blez $a0, BaseCase

  RecursiveStep:
    subi $sp,$sp,8          # Decrement the stack pointer
    sw $ra,0($sp)           # Store the return address for later
    sw $a0,4($sp)           # Store the value of n

    subi $a0,$a0,1          # Decrement n by 1
    jal factorial           # Do the recursive call

    lw $a0,4($sp)           # Restore n
    lw $ra,0($sp)           # Restore the return address
    addi $sp,$sp,8          # Restore the stack pointer

    mul $v0,$v0,$a0         # Compute n × factorial(n-1)

    jr $ra                  # Return

  BaseCase:
    addi $v0,$zero,1        # 0! = 1! = 1

    jr $ra
```

Salva dati su stack

Recupera dati da stack

# Implementazione ricorsiva (completa)

```
factorial:
  blez $a0, BaseCase

  RecursiveStep:
    subi $sp,$sp,8
    sw $ra,0($sp)
    sw $a0,4($sp)

    subi $a0,$a0,1
    jal factorial

    lw $a0,4($sp)
    lw $ra,0($sp)
    addi $sp,$sp,8

    mul $v0,$v0,$a0

    jr $ra

  BaseCase:
    addi $v0,$zero,1

    jr $ra
```

Salva dati su stack

Recupera dati da stack

# Chiamate annidate e stack

# Da ricorsione a iterazione

*È sempre possibile trasformare un problema ricorsivo in uno iterativo (e viceversa)*

**Caso semplice**, se la funzione ricorsiva ha una sola chiamata ricorsiva:

```
funzione( argomenti0 )
    if caso_base
        istruzioni del caso base
        return risultato
    else
        istr. prima della ricorsione
        calcolo di argomenti1
        funzione( argomenti1 )
        istr. dopo la ricorsione
        return risultato
```

L'esecuzione si svolgerà più o meno così:

```
funzione( argomenti0 )
    istr. prima della ricorsione
    calcolo di argomenti1
        istr. prima della ricorsione
        calcolo di argomenti2
            istr. prima della ricorsione
            calcolo di argomenti3
                ....
                istruzioni del caso base
                ....
            istr. dopo la ricorsione
        istr. dopo la ricorsione
    istr. dopo la ricorsione
```

# … versione iterativa

Due cicli eseguono nell'ordine:

- la ripetizione delle **istruzioni precedenti** alla chiamata ricorsiva (compresa di **aggiornamento degli argomenti** della chiamata),
- il **caso base**,
- e poi la ripetizione (per lo stesso numero di volte) delle **istruzioni seguenti** la chiamata ricorsiva (se necessario riottenendo il valore che avevano gli **argomenti**)

```
funzione( argomenti)
    K=0
    while (not caso base)
        K += 1
        istr. prima della ricorsione
        aggiornamento degli argomenti
    istruzioni del caso base
    for (i=0 ; i<K ; i++)
        ricostruzione degli argomenti
        istr. dopo la ricorsione
```

**NOTA** se la ricorsione è multipla, un contatore non è sufficiente per ricostruire la struttura delle chiamate. Può essere necessario usare uno stack per simulare la gestione corretta delle chiamate

# Esempio

```
def fattoriale(N):
    if N < 2:
        return 1          # caso base
    else:
        N1 = N – 1        # aggiorno args
        F1 = fattoriale( N1 )   # ricorsione


        F2 = F1 * N       # istr. dopo ric.
        return F2
```

**NOTA:** possiamo semplificare la versione iterativa usando la commutatività dell'operazione prodotto per svolgere i prodotti da N a 1 invece che da 1 a N ed eliminando il conteggio non più necessario

```
def fattoriale(N):
    i=0                   # contatore
    while (N >1):
        i = i + 1         # conto  chiamate
        N = N – 1         # aggiorno args
    F1 = 1                # caso base
    for j in range(i):
        N = N + 1         # ricostruisco args
        F1 = F1 * N       # istr. dopo ric.
    return F1
```

```
def fattoriale(N):
    F1 = 1                # caso base
    while (N >1):
        N = N – 1         # aggiorno args
        F1 = F1 * N       # istr. dopo ric.
    return F1
```