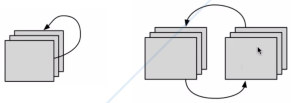
LE FUNZIONI RICORSIVE

- **funzione ricorsiva**: in c è una funzione che quando invocata richiama se stessa per completare un calcolo, esistono due tipi:

  1) **ricorsione diretta**: la ricorsione di una funzione che chiama se stessa

  2) **ricorsione indiretta**: è causata da due o più funzioni che si richiamano a vicenda



- la base matematica: ragionamenti induttivi in cui ho un **caso base P(0)** di cui conosco il risultato + un p**asso induttivo P(n+1)** risolvibile per semplificazione o dimostrazione induttiva: suppongo che il caso P(n) valga e dimostro che vale P(n+1) ;

(es. Fattoriale, i numeri pari, distributività del quadrato rispetto alla moltiplicazione $(2n)^2 = 4n^2$)

il fattoriale di un naturale N (N!):
se N=0, il fattoriale N! è 1 — caso base
se N>0 il fattoriale N! è N * (N-1)! — passo induttivo

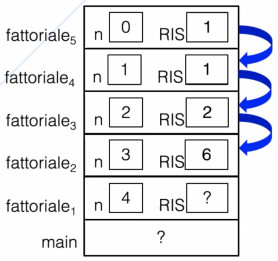
- Definire un metodo ricorsivo:
  1- deve esistere un caso base ovvero un problema di facile risoluzione il cui risultato sia noto,
  2- definisco un passo induttivo che sia riconducibile al caso base tramite scomposizione in problemi più semplici che convergono al caso base

es. calcolo di 4! = 24

```
long fattoriale (int n) {
  if (n == 0) return 1;
  return (long) n * fattoriale(n-1);
}
```

Caso base

Passo induttivo



- attivazione di funzioni ricorsive: nella pila si aggiungono diverse attivazioni dello stesso sottoprogramma, tutte sospese, che eseguono lo stesso codice e operano su copie distinte di parametri e variabili locali, e convergono al caso base. Quando il caso base viene raggiunto la funzione restituisce il risultato alla copia preedente e così via con una sequenza di restituzioni con cui si ripercorre all'indietro la sequenza delle chiamate in sospeso finchè la chiamata originaria non restituisce il risultato all'ambiente chiamante.

- errori comuni: bisogna assicurarsi che la funzione termini esplicitanto sempre almeno un caso base, assicurarsi che ogni passo induttivo converga verso il caso base

- ricorsione e iterazione possono risolvere problemi strutturati in modo simile, qualsiasi problema ricorsivo è risolvibile anche in un modo iterativo.
  vantaggi: funzioni sintetiche ed eleganti, in poche linee risolvono problemi anche abbastanza complessii
  svantaggi: numerose chiamate della funzione sulla pila che possono avere ripercussioni negativa in memoria

esempi

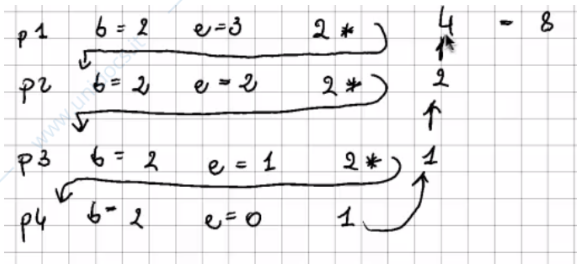
(es. funzione di calcolo potenza versione iterativa)

```
// potenza di interi con esponenti interi
// versione ITERATIVA
int potenza_it (int base, int esp) {

  int ris = 1;

  for (int i=0; i<esp; i++)
    ris *= base;

  return ris;
}
```

(es. funzione di calcolo potenza versione ricorsiva)

```
// potenza di interi con esponenti interi
// versione RICORSIVA
int potenza_ric (int base, int esp) {

  if (esp == 0) return 1;

  return base * potenza_ric(base, esp-1);
}
```

scompongo il caso di partenza in singoli casi:
dato 2^3 richiamo funzione tante volte successive e calcolo ogni volta la potenza diminuita di 1:
$2^3 >> 2^2 >> 2^1 >> 2^0 = 1 \rightarrow$ ottengo la condizione dell'if e ho return 1



(es. numeri di Fibonacci)

$F = \{f_0, \ldots, f_n, \ldots\}$

$f_0 = 1$
$f_1 = 1$
} casi base (sono 2)

Per $n > 1$, $f_n = f_{n-1} + f_{n-2}$ } 1 passo induttivo

versione iterativa:

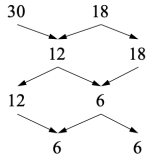
```
int fibo_it (int n) {
  if (n == 0 || n == 1) return 1;       // casi base
  int ultimo = 1, penultimo = 1, fibo;
  for (int i=2; i<=n; i++) {            // passo iterativo
    fibo = ultimo + penultimo;
    penultimo = ultimo;
    ultimo = fibo;
  }
  return fibo;
}
```

versione ricorsiva:

```
int fibo_ric (int n) {
  if (n == 0 || n == 1) return 1;       // casi base
  return fibo_ric(n-1) + fibo_ric(n-2); // passo induttivo
}
```

(es. massimo comune divisore secondo euclide)

se M=N allora MCD è N;
se M>N allora esso è il MCD tra N e M-N;
se N>M allora esso è il MCD tra M e N-M.



versione iterativa:

```
int euclide_it (int m, int n) {
  while (m != n) {
    if ( m > n )
      m = m - n;
    else
      n = n - m;
  }
  return m;
}
```

versione ricorsiva:

```
int euclide_ric (int m, int n) {
  if ( m == n )                  // caso base
    return m;
  if ( m > n )                   // passo induttivo
    return euclide_ric(m - n, n);
  else
    return euclide_ric(m, n - m);
}
```

(es. parole palindrome)

- caso base: parola di una sola lettera → palindroma
- passo induttivo: controllo figli estremi e poi controllo la sottostringa ottenuta fino ad arrivare al caso base

versione iterativa

```
int palindromo_it (char parola[]) {

  int da = 0;
  int a = strlen(parola) - 1;

  while (da < a) {
    if (parola[da] != parola[a]) return 0;
    da++;
    a--;
  }

  return 1;
}
```

versione ricorsiva:

```
int palindromo (char parola[], int da, int a) {

  if (da >= a)                    // caso base
    return 1;
  else                          // passo induttivo
    return (parola[da] == parola[a] &&
            palindromo(parola, da+1, a-1));
}
```

(es. inversione si una stringa)

- caso base: stringa vuota o con un solo carattere
- passo induttivo: inverto gli estremi e esclusi essi inverto gli estremi della sottostringa ottenuta e così via fino ad avere una stringa vuota o di un solo carattere

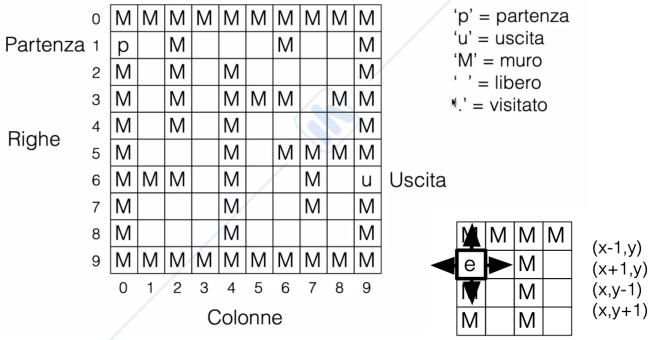
```
void inversione (char str[], int n) {

  if ( n <= 1 ) return;    // caso base

  inversione(&str[1], n-2);// passo ind. su stringa ridotta

  char temp = str[n-1];    // inversione caratteri estremi
  str[n-1] = str[0];
  str[0] = temp;
}
```

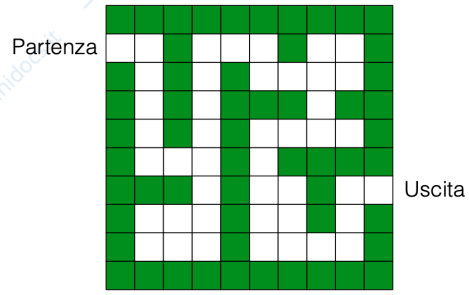
(es. labirinto)

- definisco la truttrura del labirinto, uso due coordinate x, y, per spostarmi nel labirinto

- caso base: a- posizione fuori dal labirinto, b- posizione già visitata, c- muro, d- uscita
  a- x<0 || x==DIM||y=<0 ||y==DIM;
  b- valore casella == '.' ;
  c- valore casella == 'M';
  d- valore casella == 'u';

- passo induttivo: ricerca, a partire da una casella valuto quali sono le direzioni percorribili, marco la casella come già visitata e proseguo nella direzione libera che mi porta verso l'uscita






```c
#include <stdio.h>

#define DIM 10

typedef char t_labirinto[DIM][DIM];

int trova_uscita (t_labirinto, int, int);

int main () {

  t_labirinto labirinto = {
     'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M',
     'p', ' ', 'M', ' ', ' ', ' ', 'M', ' ', ' ', 'M',
     'M', ' ', 'M', ' ', 'M', ' ', ' ', ' ', ' ', 'M',
     'M', ' ', 'M', ' ', 'M', 'M', 'M', ' ', 'M', 'M',
     'M', ' ', 'M', ' ', 'M', ' ', ' ', ' ', ' ', 'M',
     'M', ' ', ' ', ' ', 'M', ' ', 'M', 'M', 'M', 'M',
     'M', 'M', 'M', ' ', 'M', ' ', 'M', ' ', 'M', 'u',
     'M', ' ', ' ', ' ', 'M', ' ', ' ', 'M', ' ', 'M',
     'M', ' ', ' ', ' ', 'M', ' ', ' ', ' ', ' ', 'M',
     'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M', 'M'
  };

 int x, y;                              // individuazione partenza
 for (x=0; x<DIM; x++) {
   for (y=0; y<DIM; y++)
     if (labirinto[x][y] == 'p') break;
   if (y<DIM) break;                    // partenza trovata nella riga x
 }

 if (trova_uscita(labirinto, x, y)) {       // stampa risultato
   printf("Uscita trovata:\n\n");
   for (int i=0; i<DIM; i++) {
     for (int j=0; j<DIM; j++)
       printf("%c ", labirinto[i][j]);
     printf("\n");
   }
 } else printf("Nessuna uscita trovata\n\n");

 return 0;
}

 int trova_uscita (t_labirinto lab, int rig, int col) {

   // coordinate fuori dal labirinto >> non procedere
   if (rig<0 || rig==DIM || col<0 || col==DIM) return 0;

   // se la casella e' un muro o e' gia' stata visitata >> non
 procedere
   if (lab[rig][col] == 'M' || lab[rig][col] == '.') return 0;

   // se la casella e' l'uscita >> termina
   if (lab[rig][col] == 'u') return 1;

   lab[rig][col] = '.'; // marca casella corrente come visitata

   if (trova_uscita(lab, rig+1, col)) return 1;// esplora direzioni
   else if (trova_uscita(lab, rig-1, col)) return 1;
   else if (trova_uscita(lab, rig, col+1)) return 1;
   else if (trova_uscita(lab, rig, col-1)) return 1;

   lab[rig][col] = ' ';                     // cancella marcatura
   return 0;                                // scarta percorso
 }
```