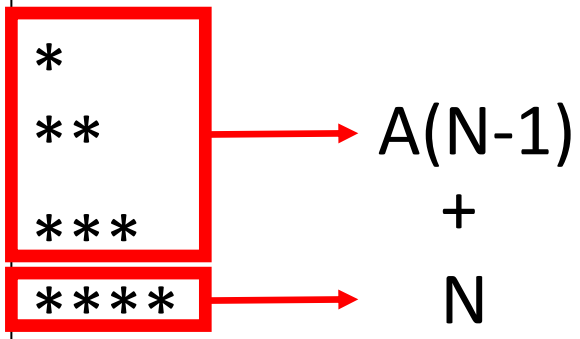
Ricorsione

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Motivazione

- La ricorsione è una tecnica di risoluzione di problemi che consiste nell'esprimere un problema in termini di sottoproblemi simili all'originale ma più semplici



A(N) = "numero totale di asterischi per disegnare un triangolo di base ed altezza N>0"

- Invece di provare a trovare una formula per A(N), posso esprimere A(N) in termini di A(N-1):

$$A(N) = N+A(N-1)$$

- La formula vale per N>1; se N=1?

$$A(1)=1$$

Motivazione

- La ricorsione è una tecnica di risoluzione di problemi che consiste nell'esprimere un problema in termini di sottoproblemi simili all'originale ma più semplici

```
*
**
***
****
```

A(N) = "numero totale di asterischi per disegnare un triangolo di base ed altezza N>0"

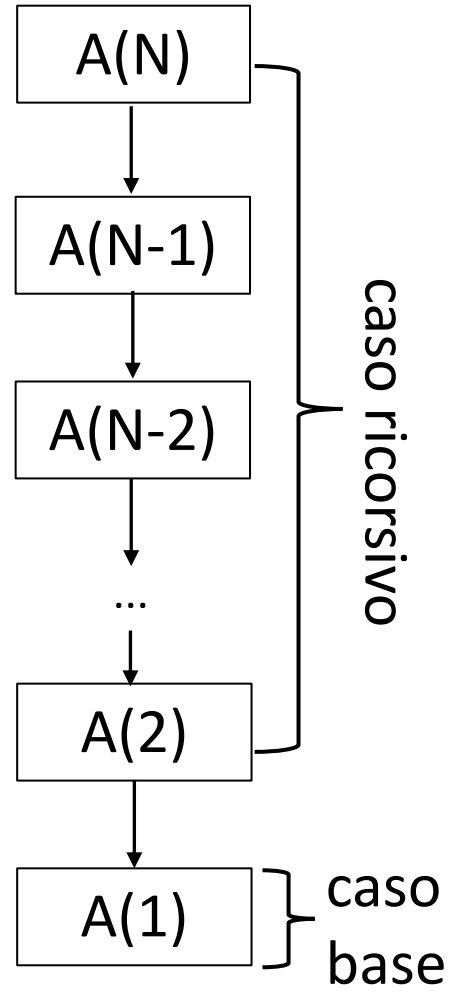
A(1)=1

A(N) = N+A(N-1)

- A(4) = 4 + A(3) =
  = 4 + 3 + A(2) =
  = 4 + 3 + 2 + A(1) =
  = 4 + 3 + 2 + 1 = 10

Motivazione

- Cosa abbiamo fatto?
- Abbiamo espresso un problema in termini di una sua versione più semplice: A(N) espresso in termini di A(N-1)
  - abbiamo identificato una gerarchia di complessità: A(N) più complesso di A(N-1)
  - possiamo calcolare la soluzione per N espressa in termini di N-1, e continuare a “semplificare il problema” finché si giunge ad un caso che sappiamo risolvere: A(1)
  - Il caso semplice che sappiamo risolvere direttamente, A(1), lo chiamiamo caso base
  - il generico A(N), N>1, lo chiamiamo caso ricorsivo



Soluzioni Ricorsive

1. Identificare il fattore da cui dipende la complessità del problema (mappabile in un numero naturale n)

2. Riconoscere tutti i casi base (se ne perdiamo uno, il programma potrebbe non terminare mai)

3. Esprimere il problema per il caso n in funzione di un caso n'<n (controllare di invocare la funzione per un'istanza più semplice del problema, ovvero che ci avvicina al caso base)
   - In altre parole, rispondo alla domanda: "se avessi la soluzione per n', come potrei calcolare la soluzione per n?"

Esempio: Somma 1+2+...+n


```c
/* PRE: n>=0; POST: restituisce 1+2+...+n */
int somma1n(int n) {
     // complessità in termini di n
     // somma(n)=n+(n-1+n-2+...+1)=somma(n)=n+somma(n-1)
     if(n==0)
          return 0;
     else
          return n+somma1n(n-1);
     //return (n==0)?0:n+somma1n(n-1);
}
```

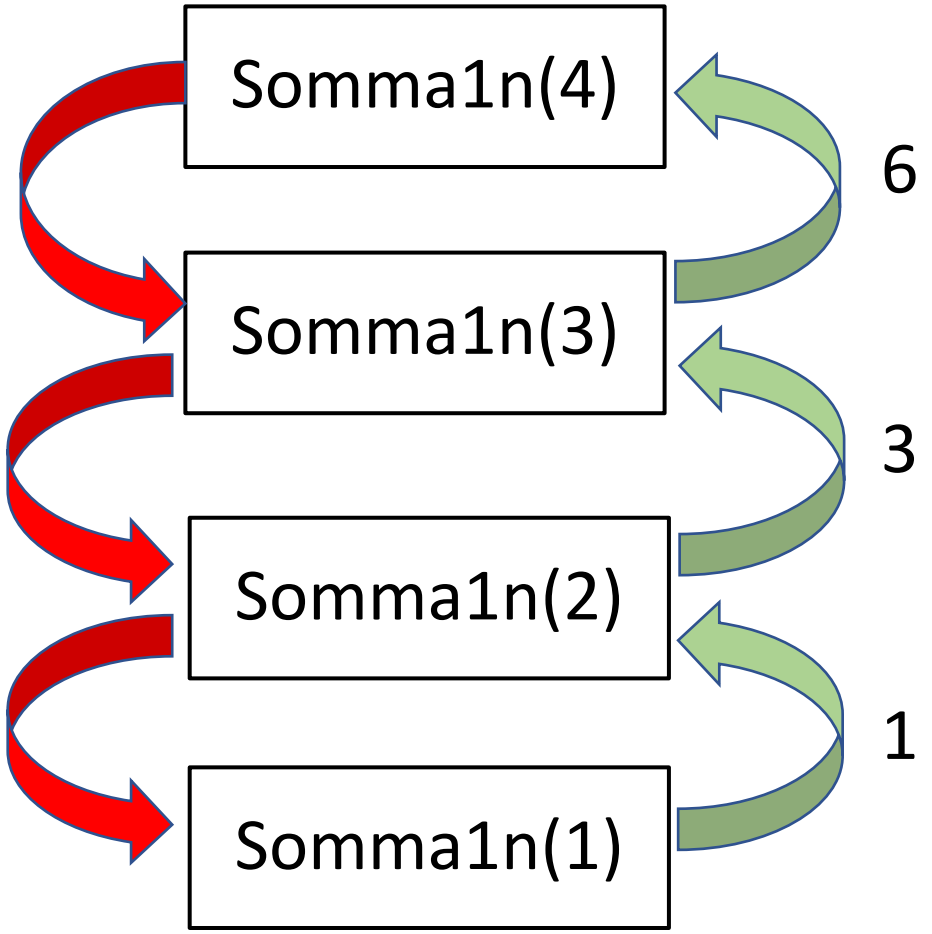
Esecuzione Somma1n()

- Somma1n(4) = 4 + Somma1n(3)

  6 + Somma1n(2)

  3 + Somma1n(1)

  1



Esempio: Potenza


```
/*
     PRE: esp>=0, base!=0;
     POST: restituisce base^esp
*/
// 2^n= 2 * 2^n-1
int potenza(int base, int esp) {
     if(esp==0)
          return 1;
     return base*potenza(base, esp-1);
}
```

Ordine Stampe


```
void f(int n) {

      printf(" ( ");
      if(n==0)
            printf(" BASE ");
      else {
            printf(" PRIMA-%d ", n);
            f(n-1);
            printf(" DOPO-%d ", n);
      }
      printf(" ) ");
}
```

Stampa:

(PRIMA-3 (PRIMA-2 (PRIMA-1 (BASE) DOPO-1) DOPO-2) DOPO-3)

Esempio: Stampa Decrescente


```c
/* PRE: n>0; POST: stampato
n
n-1
...
1
*/

void stampa_decrescente(int n) {
    if(n>0) {
        printf("%d\n", n);
        stampa_decrescente(n-1);
    }
}
```

stampa_decrescente(4)

4
3
2
1

Esempio: Stampa Crescente


```
/* PRE: n>0; POST: stampato
n
n-1
...
1
*/

void stampa_crescente(int n) {
    if(n>0) {
        stampa_crescente(n-1);
        printf("%d\n", n);
    }
}
```

stampa_decrescente(4)

1
2
3
4

Ricorsione: Esempio


```
  /*
        PRE: n>=0;  POST: restituisce fibonacci di n
   */
int fibo(int n) {
   int fibmeno2=0, fibmeno1=1;
   int fib=1;
   for(int i=2; i<n; i+=1) {
      fibmeno2 = fibmeno1;
      fibmeno1 = fib;
      fib = fibmeno2+fibmeno1;
   }
   return (n<=1)?n:fib;
}
```

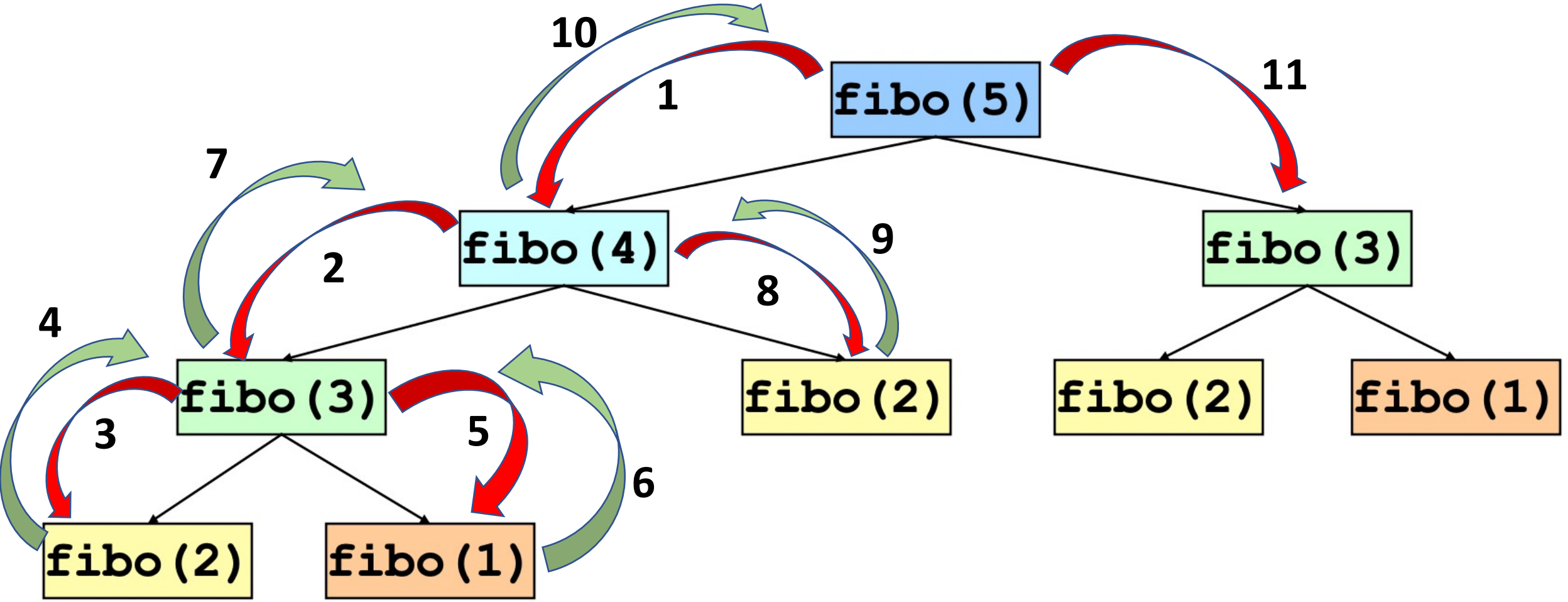
```
int fibo(int n) {
   if (n==0)
      return 0;
   if (n==1)
      return 1;
   return fibo(n-1)+fibo(n-2);
}
```

Ricorsione: Esempio


```
  /*
        PRE: n>=0;  POST: restituisce fibonacci di n
   */
int fibo(int n) {
   int fibmeno2=0, fibmeno1=1;
   int fib=1;
   for(int i=2; i<n; i+=1) {
      fibmeno2 = fibmeno1;
      fibmeno1 = fib;
      fib = fibmeno2+fibmeno1;
   }
   return (n<=1)?n:fib;
}
```

```
int fibo(int n) {
   if (n==0)
      return 0;
   if (n==1 || n==2)
      return 1;
   return fibo(n-1)+fibo(n-2);
}
```

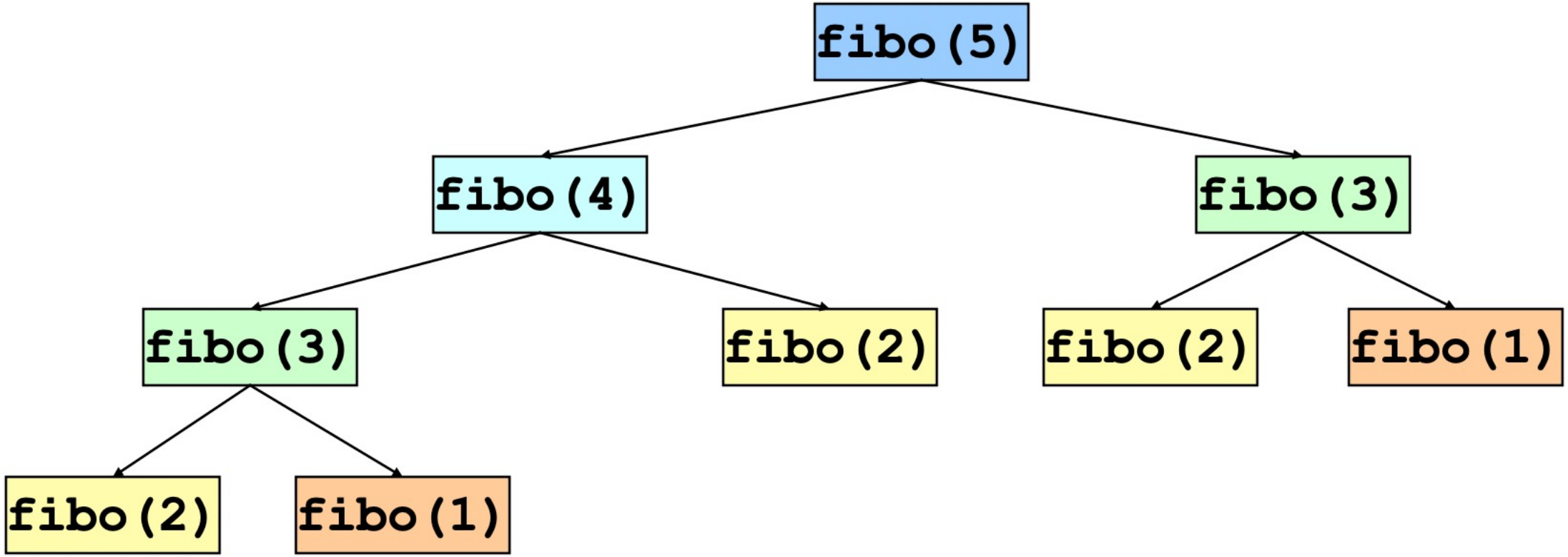
Ricorsione: ordine di visita - depth first



- I numeri indicano l'ordine di esecuzione delle funzioni

Ricorsione: Quando Non Utilizzarla

- La nostra funzione fibo ripete gli stessi calcoli più volte → inefficiente



Ricorsione: Discussione

- Per alcuni problemi fornisce una soluzione elegante e facile da leggere (ma esiste sempre una soluzione iterativa equivalente allo stesso problema)
- La ricorsione richiede tempo aggiuntivo per la gestione dello stack
- Consuma più memoria di una versione iterativa
  - alloca un nuovo stack frame a ogni chiamata, definendo una nuova ulteriore istanza delle variabili locali e dei parametri ogni volta

Perché la Ricorsione “Funziona”?

Vedere Diapositive sulla Correttezza

Esempio: Trova Carattere


```
/*
      PRE: s è un puntatore a stringa
      POST: restituisce 1 se x è presente in s
                        0 altrimenti
*/
int trova_char(char *s, char x) {
 // complessità in termini della lunghezza della stringa in input
 // ciao = c + iao -> c è il carattere che cerco oppure l'ho
 // trovato nel resto della stringa?
      if (*s=='\0')
            return 0;
      else
            return (*s==x) || trova_char(s+1, x);
}
```

Esempio: Massimo Array


```
/*
     PRE: A è un array di dim>0 elementi
     POST: restituisce il valore massimo in A
*/
int max_array(int *A, int dim) {
     // complessità in termini di dim (lunghezza array)
     if (dim==1)
          return A[0];
     int max = max_array(A+1, dim-1);
     return A[0]>max? A[0]: max;
}
```

Esempio: Array Palindromo


```
/*
     PRE: A è un array di dim>0 elementi
     POST: restituisce 1 se A è palindromo
                       0 altrimenti
*/
int array_palindromo(int *A, int dim) {
// complessità in termini della lunghezza dell'array: dim
// { x1, x2, x3, x4, x5, x6 } -> x1=x6 && {x2, x3, x4, x5} palindromo
     if (dim<=1)
          return 1;
     return (A[0]==A[dim-1]) && array_palindromo(A+1, dim-2);
}
```

Esempio: Array Palindromo


```
/*
      PRE: A è un array di dim>0 elementi
      POST: restituisce 1 se A è palindromo
                        0 altrimenti
*/
int array_palindromo(int *A, int dim) {
// complessità in termini della lunghezza dell'array: dim
// { x1, x2, x3, x4, x5, x6 } -> x1=x6 && {x2, x3, x4, x5} palindromo
      if (dim<=1)
            return 1;
      if (A[0]!=A[dim-1])
            return 0;
      return array_palindromo(A+1, dim-2);
}
```

Array Palindromo: Correttezza


```
/*
   PRE: A è un array di dim>0 elementi
   POST: restituisce 1 se A è palindromo
                     0 altrimenti
*/
int array_palindromo(int *A, int dim) {
      if (dim<=1)
            return 1;
      if (A[0]!=A[dim-1])
            return 0;
      return array_palindromo(A+1, dim-2);
}
```

caso base: per dim<=1 si restituisce 1 → corretto

A = {A[0],A[1:dim-2],A[dim-1]}
hp induttiva: array_palindromo(A+1, dim-2) restituisce 1 se A[1:dim-2] è palindromo, 0 altrimenti

Caso ricorsivo: se A[0]!=A[dim-1] si restituisce 0; altrimenti si restituisce array_palindromo(A[1:dim-2]), che è corretto per hp induttiva.

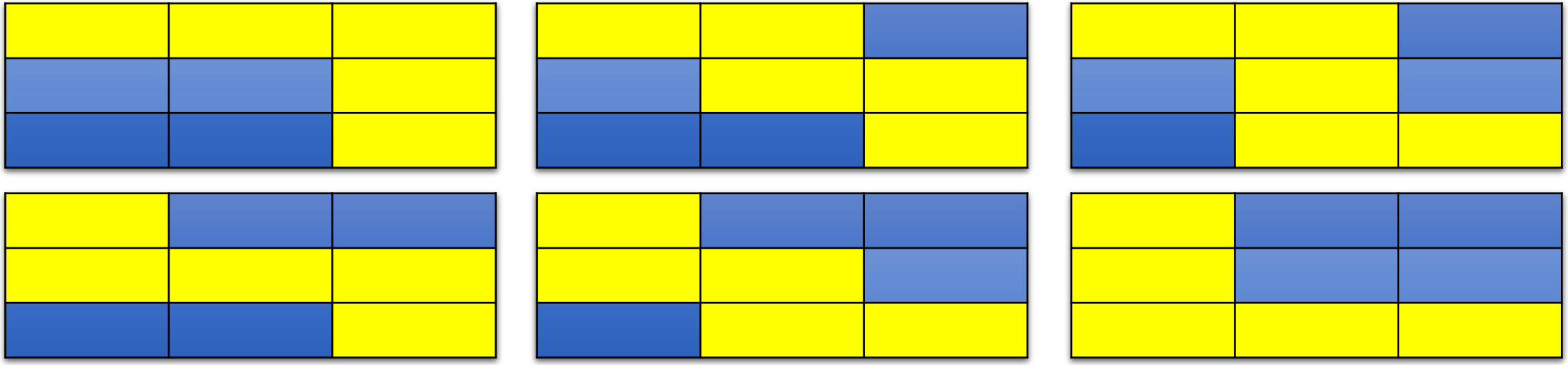
Esempio: Percorsi su griglia

- Data una griglia bidimensionale, calcolare il numero dei percorsi diversi dall'angolo in alto a sinistra a quello in basso a destra
- Le uniche mosse possibili sono:
  - spostarsi di una casella a destra
  - spostarsi di una casella in basso

sono qui	mossa possibile	
mossa possibile		
		arrivo

Esempio: Percorsi su griglia

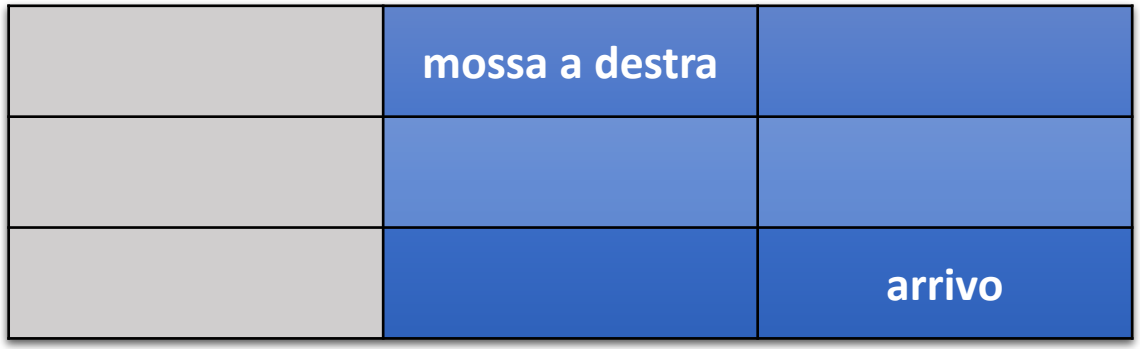
- Data una griglia bidimensionale, calcolare il numero dei percorsi diversi dall'angolo in alto a sinistra a quello in basso a destra
- Le uniche mosse possibili sono:
  - spostarsi di una casella a destra
  - spostarsi di una casella in basso
- Esempi di soluzioni:



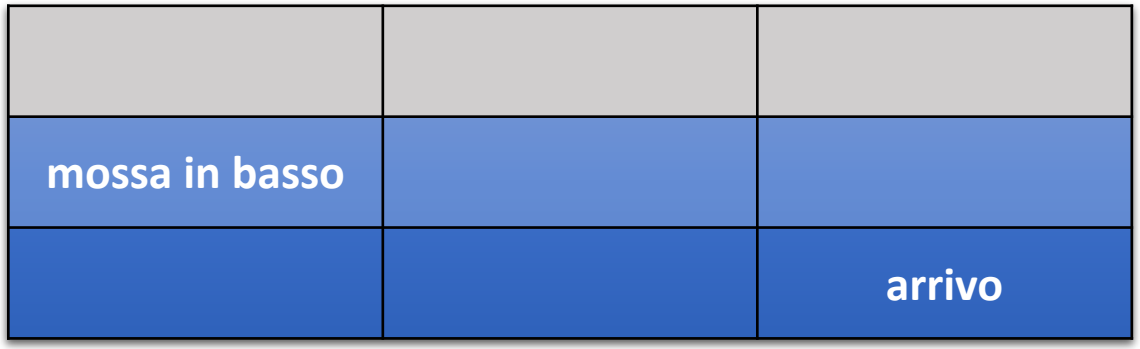
Esempio: Percorsi su griglia

3x3=9

3x2=6



2x3=6



- dopo ogni mossa diminuiamo la dimensione della griglia, che perciò può rappresentare la dimensione del nostro problema:
  - da 3*3=9 a 2*3=6 o 3*2=6

Esempio: Percorsi su griglia

- P(n,m) = griglia nxm, numero di percorsi da (0,0) -angolo in alto a sinistra- a (n-1, m-1) -angolo in basso a destra-
- Casi base:
  - Griglia 1x1: P(1,1)=1
  - P(1,y)=P(x,1)=1
  - P(0,y)=P(x,0)=0 //se una dimensione è zero, non ci sono percorsi

Esempio: Percorsi su griglia

- Da (n,m) posso spostarmi
  - a destra →(n,m-1)
  - in basso →(n-1,m)

sono qui		
		arrivo

- nell'esempio se mi muovo in basso, il numero di percorsi è sempre 1
- Quindi P(n,m) = P(n-1,m)+P(n,m-1)

3+3 percorsi		
		arrivo

P(n,m-1)

	3 percorsi	
		arrivo

P(n-1,m)

3 percorsi		
		arrivo

Casi base Ridondanti

- P(n,m) = P(n-1,m)+P(n,m-1)
- Casi base:
  - Griglia 1x1: P(1,1)=1
  - P(1,y)=P(x,1)=1
  - P(0,y)=P(x,0)=0 //se una dimensione è zero, non ci sono percorsi
- P(1,y)=P(x,1)=1 sono ridondanti:

P(1,3)=P(0,3)+P(1,2)=

0 +P(0,2)+P(1,1)=

0 + 0 + 1 = 1

Esempio: Percorsi Griglia


```
int percorsi_griglia(int righe, int colonne) {

      if (righe<=0 || colonne<=0)
            return 0;
      if (righe==1 && colonne==1)
            return 1;
      return percorsi_griglia(righe-1, colonne) +
             percorsi_griglia(righe, colonne-1);

}
```

Esercizio

Progettare ed implementare le seguenti funzioni ricorsive e dimostratene la correttezza

1. int array_pari(int X[], int dim)

   /* POST restituisce 1 se tutti gli elementi di X sono pari; 0 altrimenti */

1. int conta_occorrenze(int X[], int dim, int n)
   /* POST restituisce il numero di occorrenze di n in X */

Esercizio: Conta Occorrenze

```
/* PRE: X ha dim elementi
    POST: restituisce il numero di occorrenze di n in X
*/

int conta_occ(int X[], int dim, int n) {
  if(dim==0)  {
            return 0;
  }
   return (X[0]==n) + conta_occ(X+1,dim-1, n);
}
```

Caso Base: dim=0 restituisce 0 → corretto

X={X[0], *(X+1), ...,*(X+dim-1)}

hp induttiva: conta_occ(X+1,dim-1,n) = |{i: 1<=i<dim-1, X[i]=n}| = k

Caso Ricorsivo: se ho k istanze di n in X[1:dim-1], allora ne ho k o k+1 in X a seconda che X[0]!=n o X[0]=n.

Esercizi

Definire una funzione ricorsiva che determini se esiste un percorso che permetta di attraversare un campo fiorito, dal basso verso l'alto, senza calpestare alcun fiore.

Il campo è rappresentato da una matrice, i cui valori rappresentano la presenza di un fiore (0) oppure la sua assenza (1). La posizione iniziale è fornita come parametro della funzione.

E' possibile muoversi una casella in alto oppure una casella verso destra.

{0,0,0,1,0},

{0,1,0,1,0},

{1,0,0,1,0},

{1,0,1,1,1},

{1,0,1,0,0}

Esercizio


```
int mossa(int dim_x, int dim_y, int campo[dim_x][dim_y], int pos_x, int pos_y) {

   if ( (pos_x<0) || (pos_y<0) || (pos_y>=dim_y) || (pos_x>=dim_x) ) {
         return 0; //mossa fuori dal percorso
   }
   if (campo[pos_x][pos_y]==0)
      return 0; // calpestato un fiore, percorso non valido
   if (pos_x==0)
      return 1; // sono arrivato nella prima riga, percorso valido
   return (                                                  //muovo di una casella
      (mossa(dim_x, dim_y, campo, pos_x-1, pos_y)) ||     // in alto
      (mossa(dim_x, dim_y, campo, pos_x, pos_y+1))       // a destra
   );
}
```

Esercizi

- L'esercizio su campo fiorito, sarebbe stato possibile risolverlo con una funzione ricorsiva se le mosse possibili fossero state 3: muovi di una casella a sinistra, muovi di una casella in alto, muovi di una casella a destra? Perché?

{0,0,0,1,0},

{0,1,0,1,0},

{1,0,0,1,0},

{1,0,1,1,1},

{1,0,1,0,0}

Esercizio


```
int mossa(int dim_x, int dim_y, int campo[dim_x][dim_y], int pos_x, int pos_y) {

   if ( (pos_x<0) || (pos_y<0) || (pos_y>=dim_y) || (pos_x>=dim_x) ) {
         return 0; //mossa fuori dal percorso
   }
   if (campo[pos_x][pos_y]==0)
      return 0; // calpestato un fiore, percorso non valido
   if (pos_x==0)
      return 1; // sono arrivato nella prima riga, percorso valido
   return (                                                  //muovo di una casella
      (mossa(dim_x, dim_y, campo, pos_x-1, pos_y)) ||     // in alto
      (mossa(dim_x, dim_y, campo, pos_x, pos_y-1)) ||     // a sinistra
      (mossa(dim_x, dim_y, campo, pos_x, pos_y+1))        // a destra
   );
}
```

Percorso:

{0,0,0,1,0},

{0,1,0,1,0},

{1,0,0,1,0},

{1,0,1,1,1},

{1,0,1,0,0}

mossa(3,3) ha successo, perché?

Ricorsione -> Iterazione: Esempio

```
int confronta_array(int *X, int *Y, int dim) {
  if (dim==0)
          return 1;
  else {
    if (X[0]!=Y[0])
        return 0;
    else {
        return confronta_array(X+1, Y+1, dim-1);
    }
  }
}
```

```
int confronta_array(int *X, int *Y, int dim) {


while (dim!=0) {
   if (X[dim]!=Y[dim])
        return 0;
   dim = dim-1;
}
  return 1;
}
```

Trasformazione da Ricorsione a Iterazione

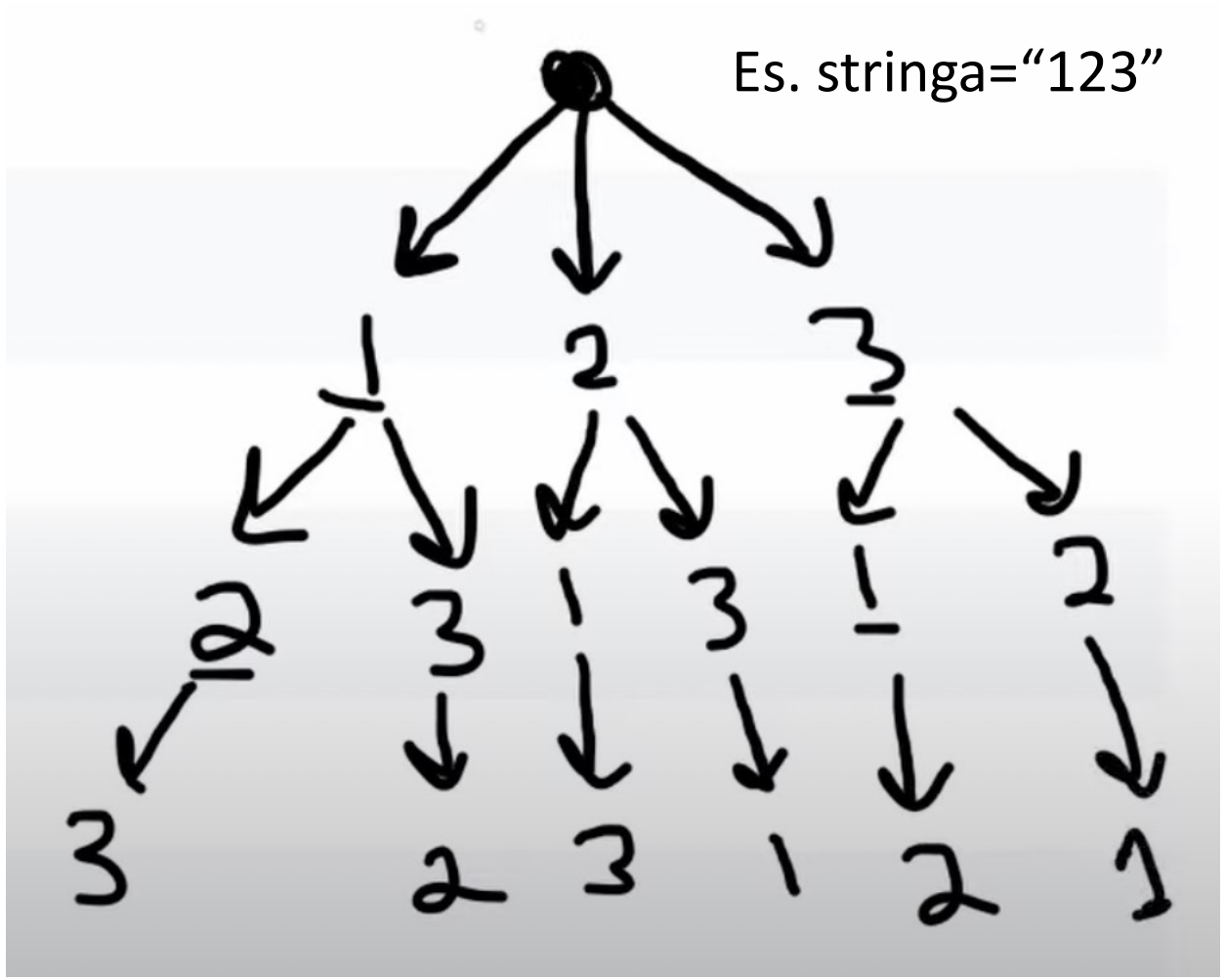

```
tipo F_ric(tipo x) {

   if (casobase(x)) {
         istruzioni_casobase;
         return risultato;
   } else {
         istruzioni_nonbase;
         return F_ric(riduciComplessita(x));
   }
}
```

```
tipo F_iter(tipo x) {
while (!casobase(x)) {
         istruzioni_non_base;
         x = riduci_complessità(x);
}
istruzioni_casobase;
return caso_base;
```

Permutazioni

- Stampare tutte le permutazioni dei caratteri di una stringa
- assumendo che i caratteri siano tutti diversi, abbiamo n*(n-1)*...*1 stringhe da stampare
- ogni sottoalbero (sottostringa) risolve un problema più semplice
- Idea: estraiamo un carattere, lo mettiamo all'inizio della stringa, e poi chiamiamo la funzione ricorsivamente per la sottostringa rimanente



Permuta Caratteri Stringa


```
void permuta_ric(char *s, char *s_inizio) {

        if (*s=='\0')
                printf("%s\n", s_inizio);
        for(int i=0, l=lung_stringa(s); i<l; i+=1) {
                scambia(s, s+i);
                permuta_ric(s+1, s_inizio); //genera tutte le permutazioni di s[1:]
                scambia(s+i, s); //ricompone la stringa originale
        }

}
/* PRE: s puntatore a stringa; POST: stampate a video tutte le permutazioni di s. */
void permuta(char *s) {
        permuta_ric(s,s);
}
```

Correttezza Funzioni Ricorsive

- Data la seguente funzione, scrivere PRE e POST e discuterne la correttezza

```
int f(int n, int m) {
    if(n==m)
        return n;
    else
        return n*f(n-1, m);
}
```

Correttezza Funzioni Ricorsive

- POST: Restituisce n*(n-1)*...*m
- PRE: n>=m>=1

- complessità in termini di n-m
- se n<m la funzione non termina mai!

```
int f(int n, int m) {
    if(n==m)
        return n;
    else
        return n*f(n-1, m);
}
```

Correttezza Funzioni Ricorsive

- POST: Restituisce n*(n-1)*...*m
- PRE: n>=m>=1
- complessità espressa in termini di n-m
- caso base n==m →POST: restituisce n⇔f(n,n)
- hp induttiva: f(n-1, m)=(n-1)*(n-2)*...*m
- f(n,m)=n*f(n-1, m)⇔n*(n-1)*(n-2)*...*m→POST verificata

```
int f(int n, int m) {
    if(n==m)
        return n;
    else
        return n*f(n-1, m);
}
```

Aritmetica dei Puntatori: Esercizi

- Date le seguenti matrici, trasformare l'operazione a[i][j] nell'operazione corrispondente che utilizza l'aritmetica dei puntatori: *(a + ...)

1. int a[4][5]; a[2][1] → *(a+ ...)
2. char b[3][2]; b[1][2] → *(b+ ...)
3. char c[2][4][3]; c[1][2][2] → *(c+ ...)

Artimetica dei Puntatori: Esercizi

- Date le seguenti matrici, trasformare l'operazione a[i][j] nell'operazione corrispondente che utilizza l'aritmetica dei puntatori: *(a + ...)

1. int a[4][5]; a[2][1] →*(a+2*5+1)
2. char b[3][2]; b[1][2] → *(b+1*2+2) →l'array ha 2 colonne, non esiste l'elemento di colonna 2
3. char c[2][4][3]; c[1][2][2] → *(c+1*4*3+2*3+2)

Esercizio

- Completare la seguente funzione ricorsiva

// date n persone in una stanza che stanno salutandosi, calcolare il numero totale di strette di mano

```
unsigned int handshake(unsigned int n) {
        //POST Restituisce il numero totale di strette di mano tra n persone
}
```