POLITECNICO
DI TORINO

Dipartimento
di Automatica e Informatica

Ricorsione e paradigma divide et impera

Paolo Camurati

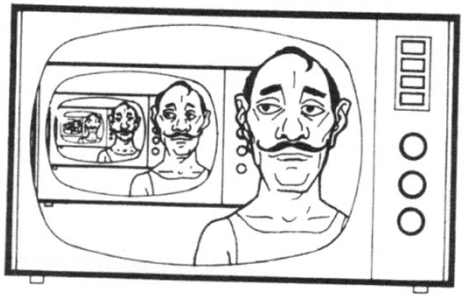
Definizione

Funzione *ricorsiva*:

- all'interno della propria definizione chiamata alla funzione stessa (*ricorsione diretta*)
- chiamata ad almeno una funzione la quale, direttamente o indirettamente, chiama la funzione stessa *(ricorsione indiretta)*

Algoritmo *ricorsivo*: si basa su funzioni ricorsive.

La soluzione di un problema S applicato ai dati D è ricorsiva se si può esprimere come:



$S(D) = f(S(D'))$ $\quad D \mathrel{!}= D_0$

$S(D_0) = S_0$

D' più semplice di D

condizione di terminazione

Motivazioni

- Natura di molti problemi:
  - risoluzione di sotto-problemi analoghi a quello di partenza (ma più piccoli)
  - combinazione di soluzioni parziali nella soluzione del problema originario
  - ricorsione come base del paradigma di problem-solving noto come **divide et impera**.
- Eleganza matematica della soluzione.

Condizione di terminazione

Ogni algoritmo deve terminare $\Rightarrow$ ricorsione finita.

Sottoproblemi semplici e risolvibili:

- banali (es.: insiemi di 1 solo elemento)
- esaurimento delle scelte lecite (es.: nel grafo è terminate la lista delle adiacenze).

Il Paradigma Divide et Impera

Divide

- da problema di dimensione n in a≥1 *problemi indipendenti e di ugual natura* di dimensione n'< n

Impera

- risoluzione di problema elementare (condizione di terminazione)

Combina

- ricostruzione di soluzione complessiva combinando le soluzioni parziali.

Risolvi(Problema):

- Se il problema è elementare:
  - Return Soluzione = Risolvi_banale(Problema)
- Altrimenti:
  - Sottoproblema$_{1,2,3,\ldots,a}$ = Dividi(Problema) ;
  - Per ciascun Sottoproblema$_i$:
    - Sottosoluzione$_i$ = Risolvi(Sottoproblema$_i$) ;
  - Return Soluzione = Combina(Sottosoluzione$_{1,2,3,\ldots,a}$) ;

condizione di terminazione

"a" sottoproblemi, ciascuno più piccolo del problema originale

chiamata ricorsiva

Valori di a

- a=1: ricorsione lineare
- a>1: ricorsione multi-via

Valori di n': ad ogni passo la dimensione si riduce di:

- un valore costante, non sempre uguale per tutti i sottoproblemi
- un fattore costante, in generale lo stesso per tutti i sottoproblemi
- una quantità variabile, sovente difficile da stimare.

Terminologia incontrata in letteratura:

- Divide and conquer: fattore di riduzione in generale costante
- Decrease and conquer: valore di riduzione in generale costante.

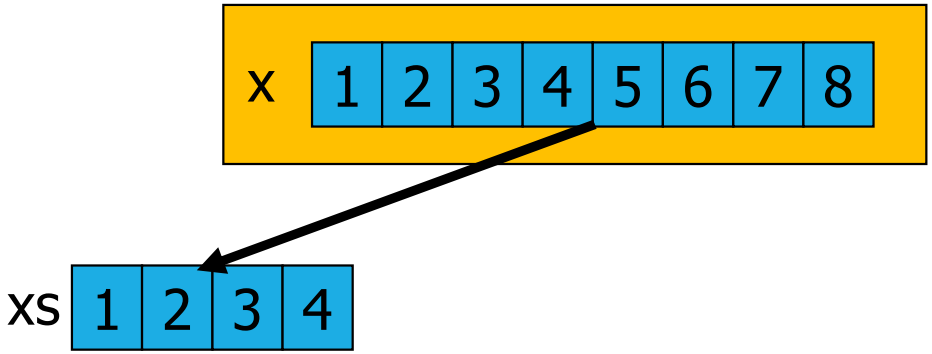
L’albero della ricorsione

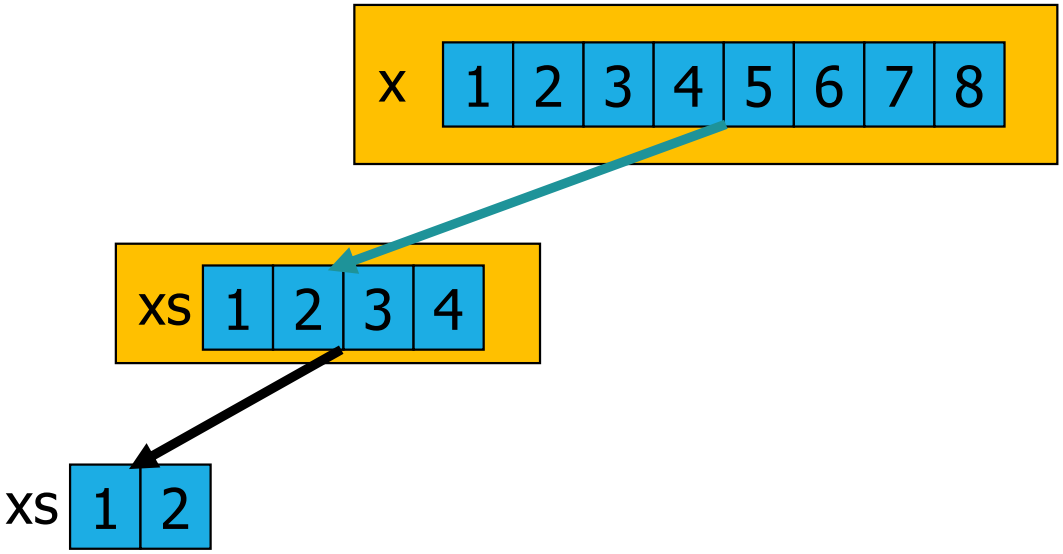
divide and conquer
a = 2 b = 2

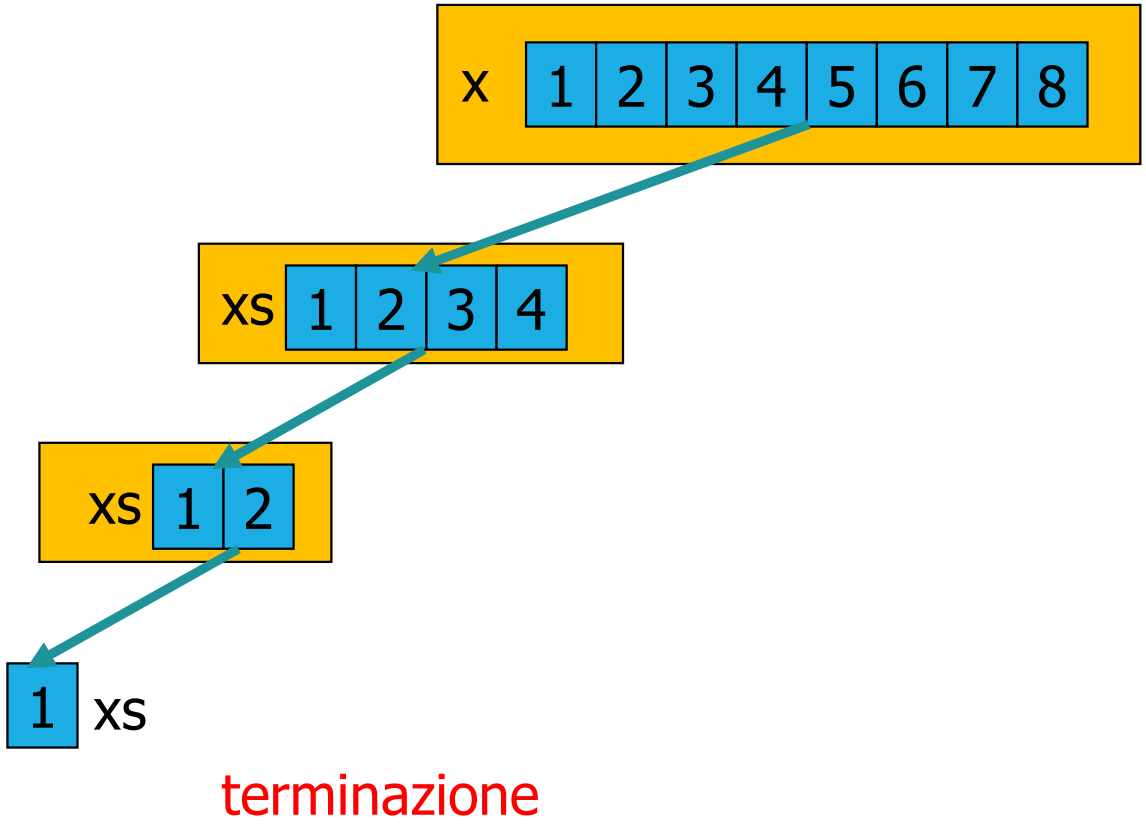
Esempio 1:

dato un vettore di $n=2^k$ interi, suddividerlo ricorsivamente in sottovettori di dimensione metà, fino alla condizione di terminazione (sottovettore di 1 sola cella).

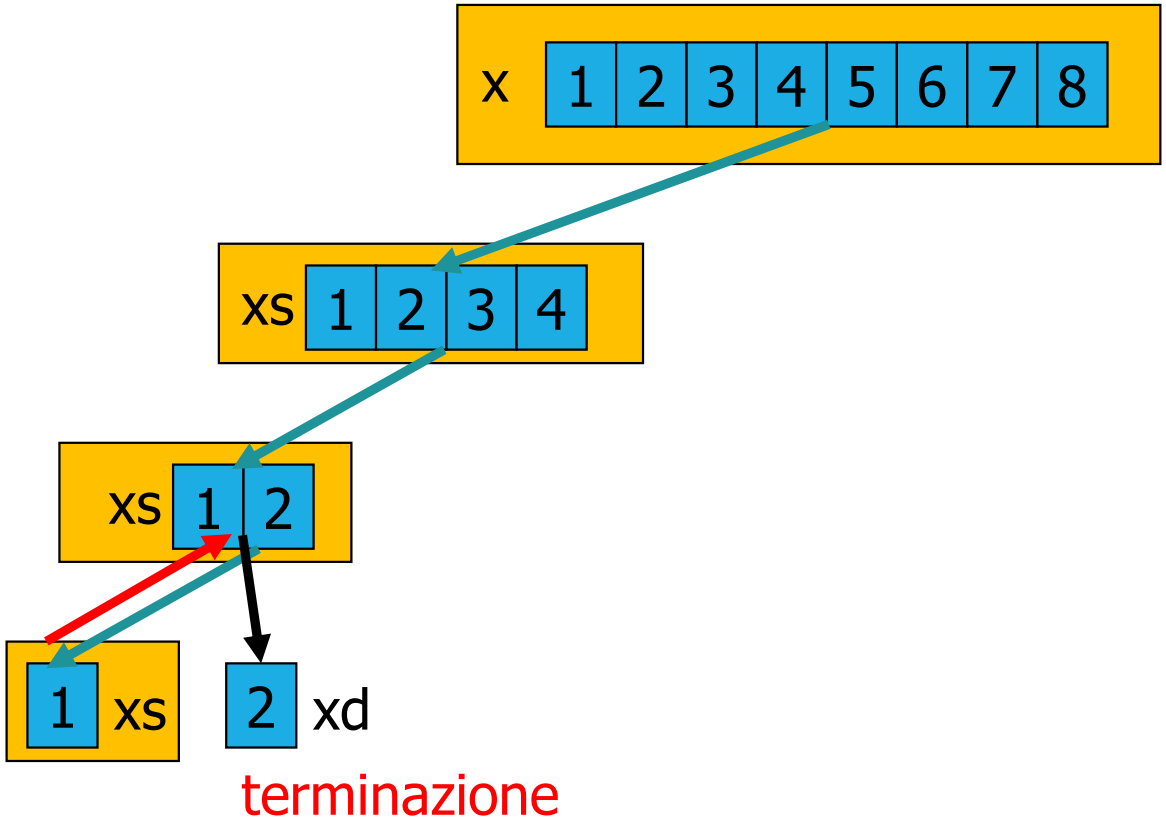

x
1
2
3
4
5
6
7
8

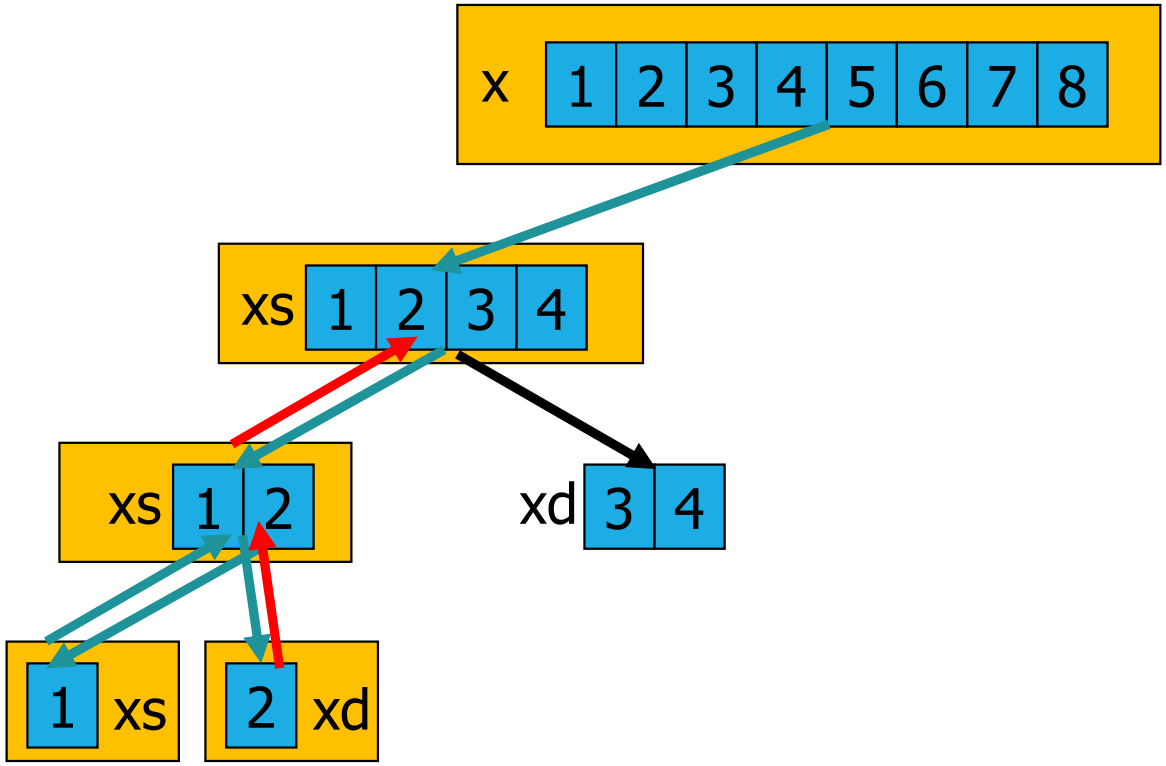

x
1
2
3
4
5
6
7
8
xs
1
2
3
4

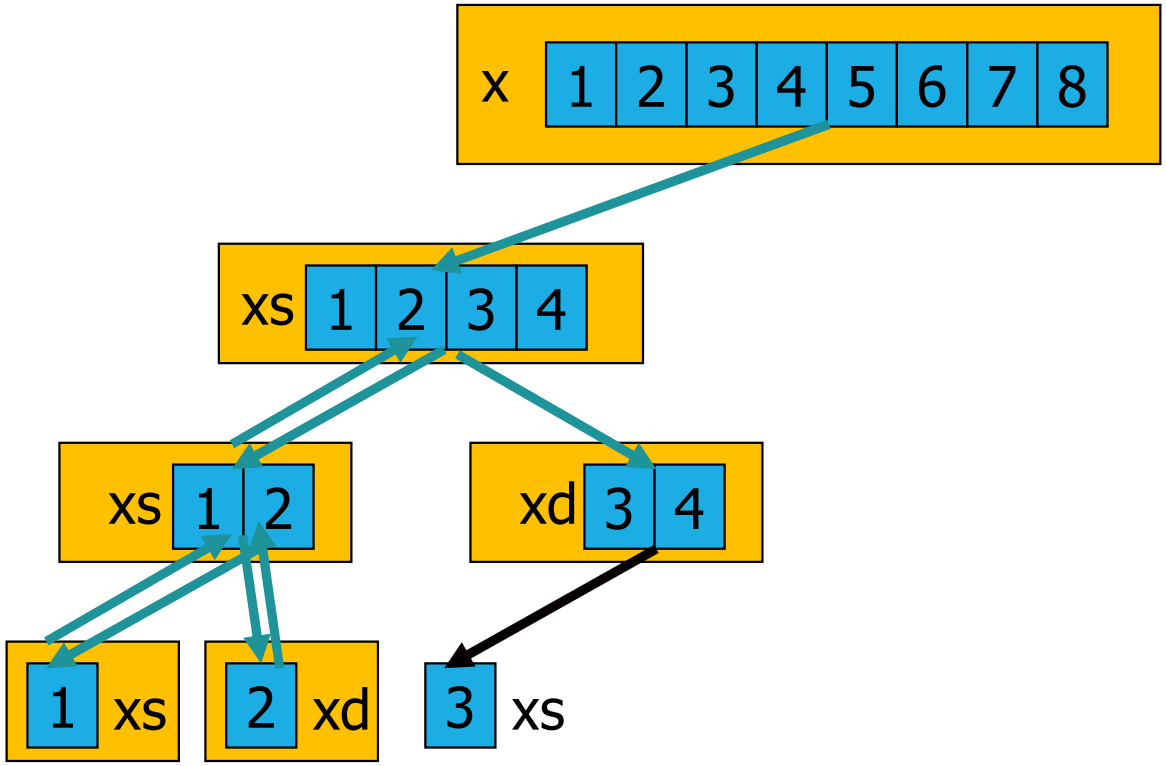

x
1
2
3
4
5
6
7
8
xs
1
2
3
4
xs
1
2



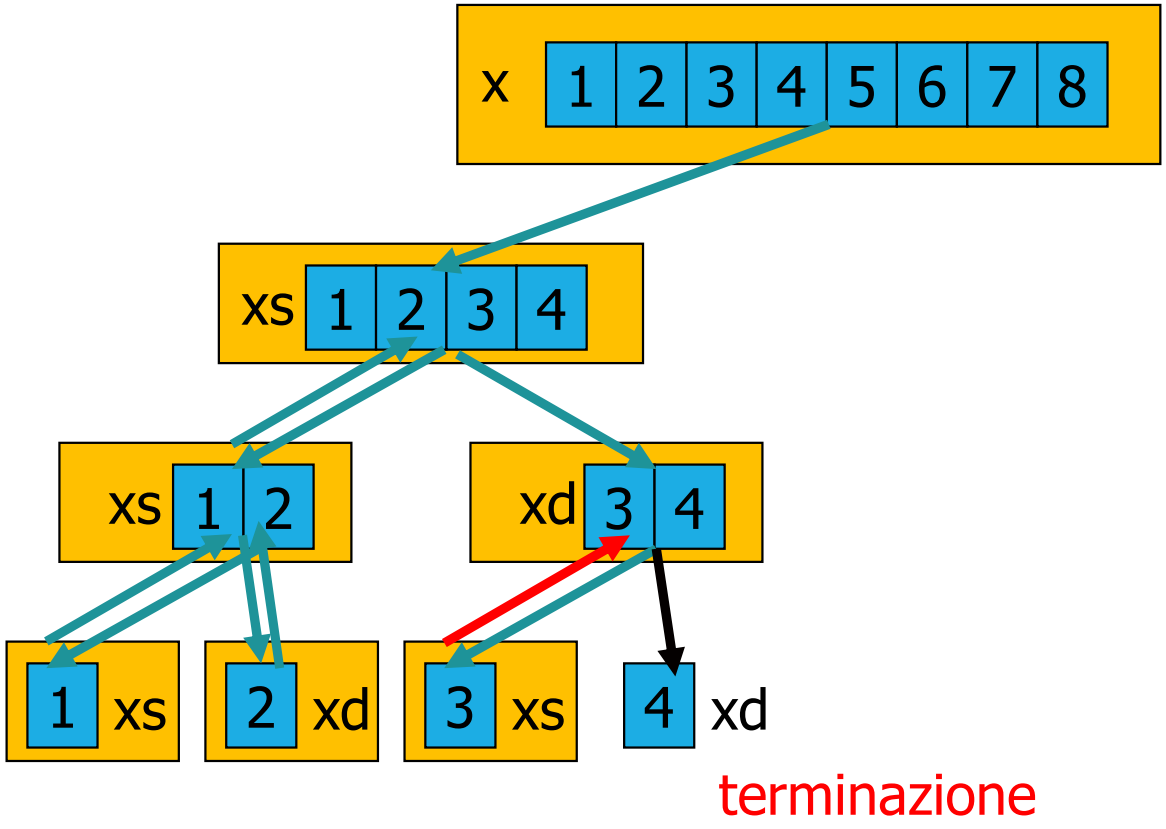
terminazione

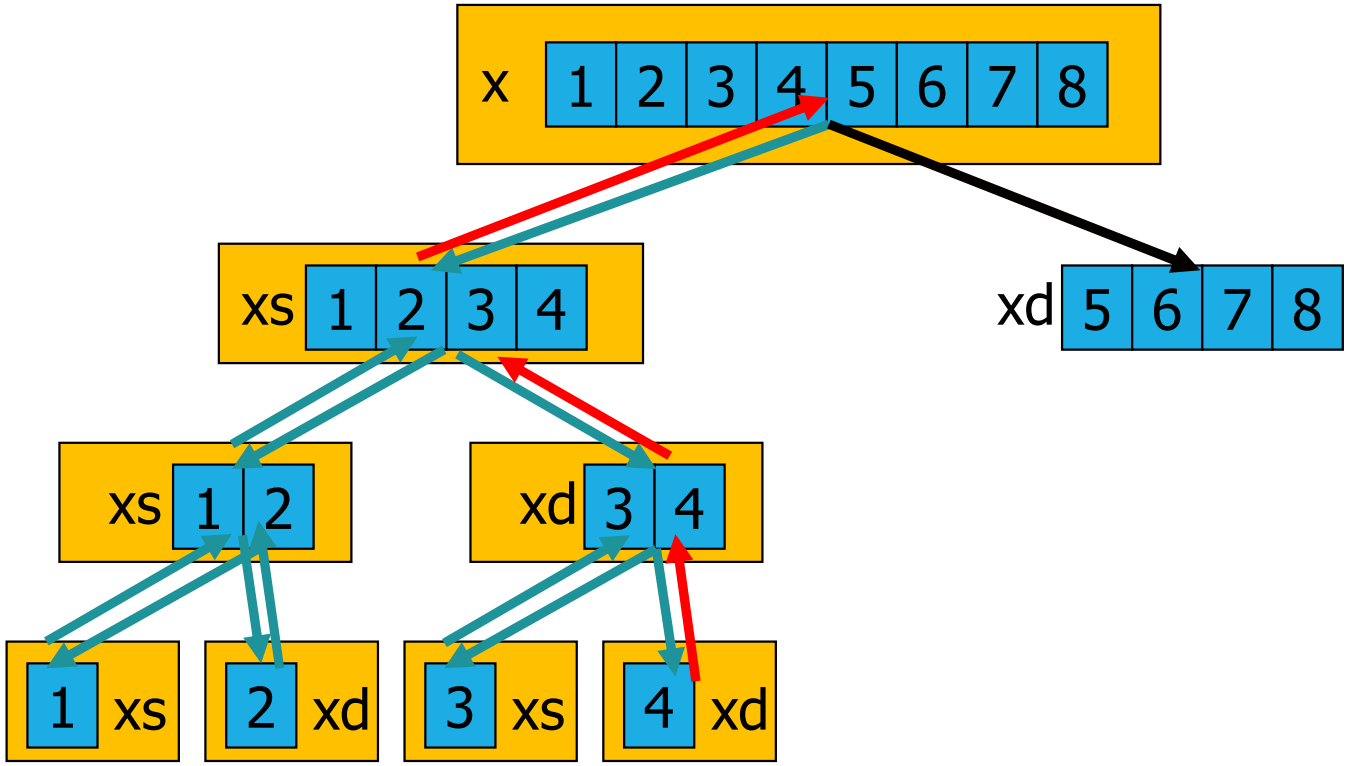

x
1
2
3
4
5
6
7
8
xs
1
2
3
4
xs
1
2
1
xs
2
xd
terminazione

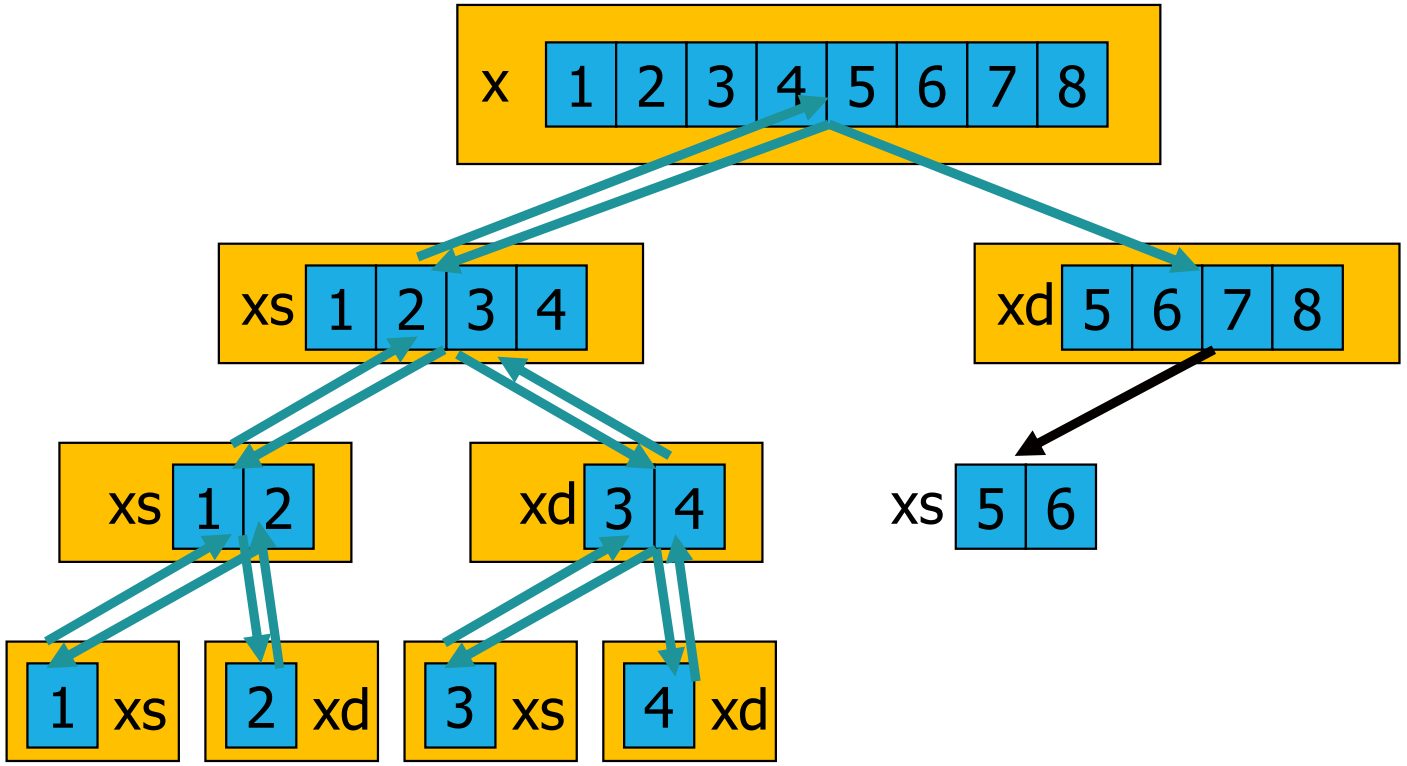

x
1
2
3
4
5
6
7
8
xs
1
2
3
4
xs
1
2
xd
3
4
1
xs
2
xd

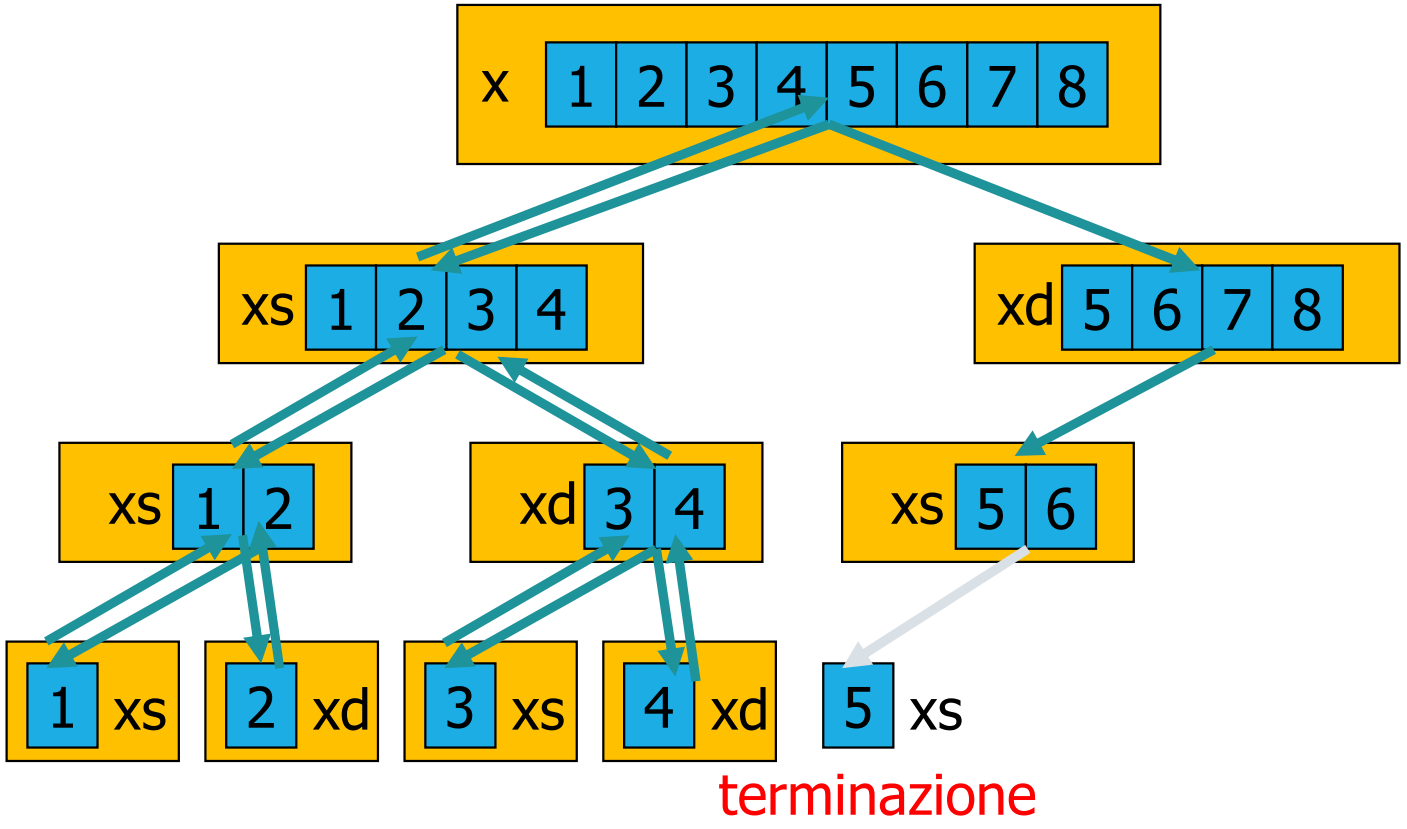


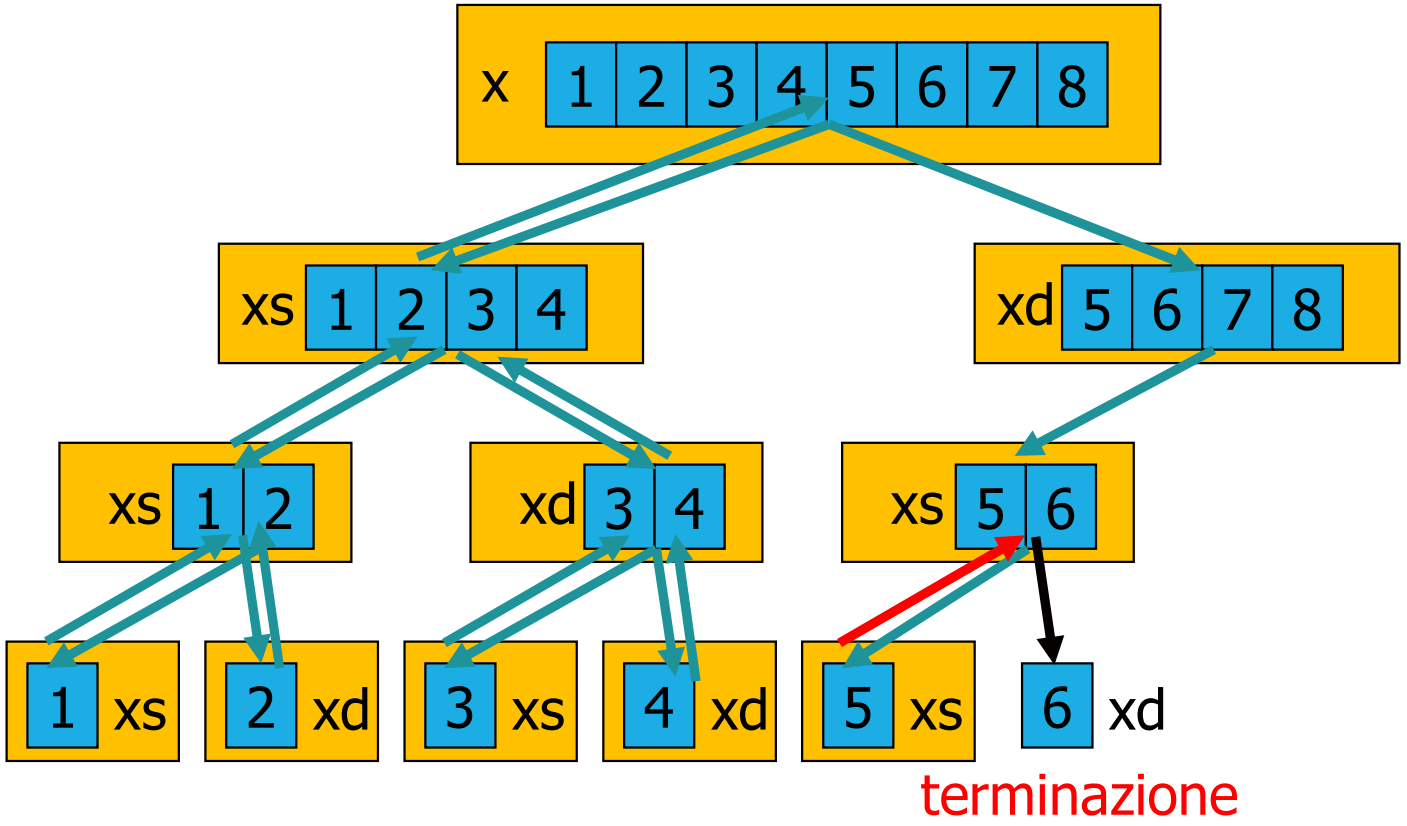
terminazione

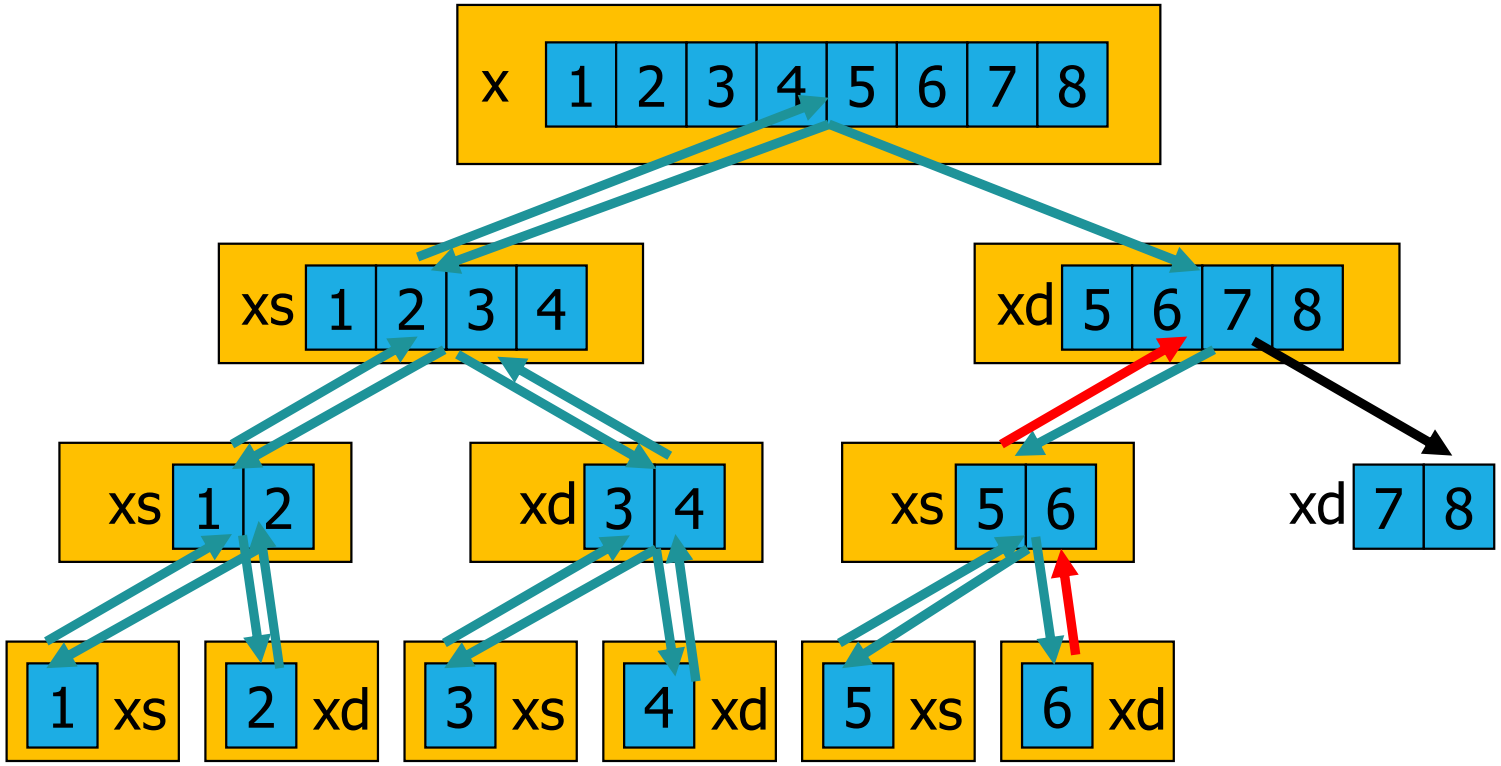

x
1 2 3 4 5 6 7 8
xs
1 2 3 4
xs
1 2
xd
3 4
1 xs
2 xd
3 xs
4 xd
terminazione

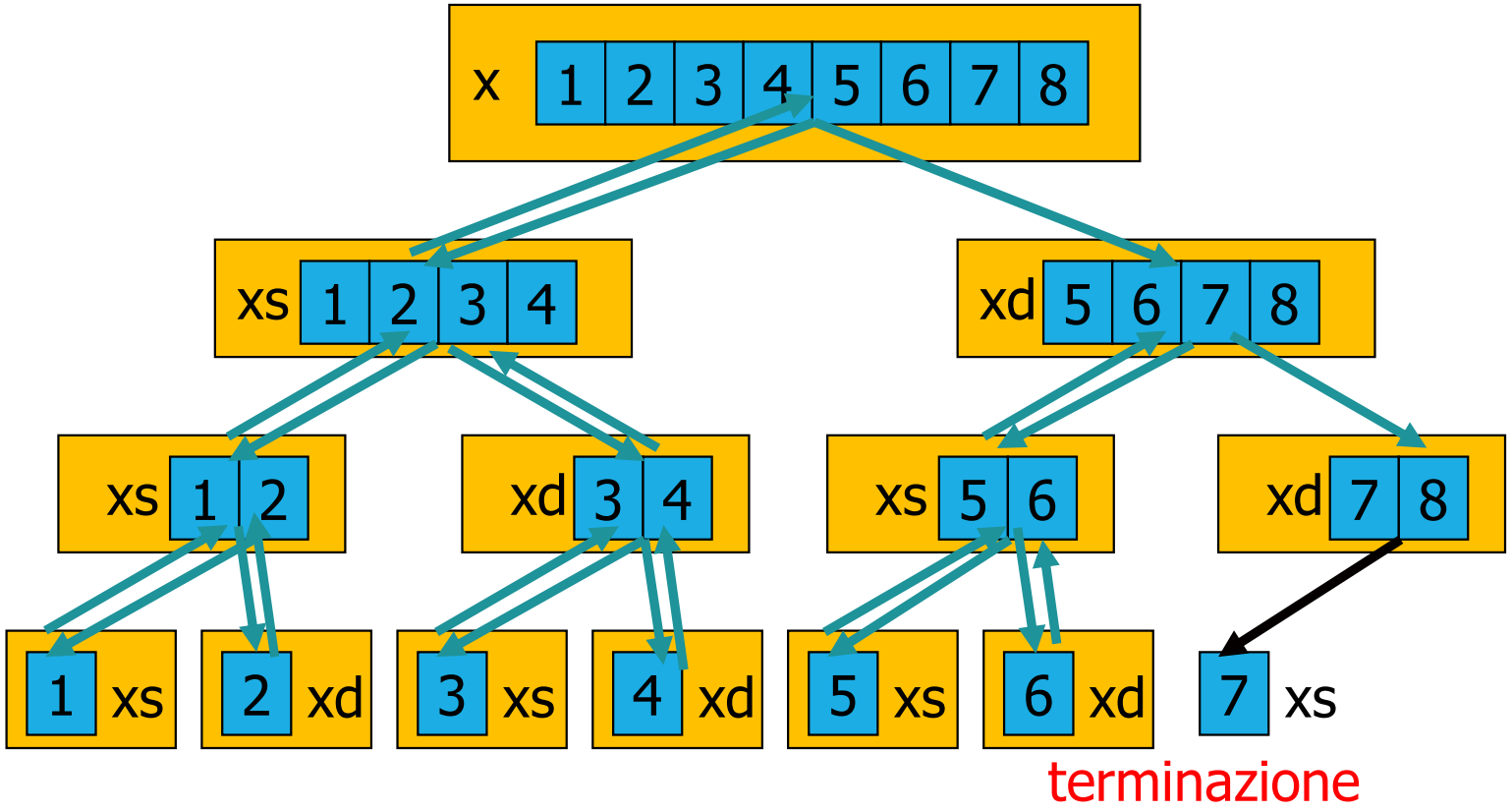

x
1
2
3
4
5
6
7
8
xs
1
2
3
4
xd
5
6
7
8
xs
1
2
xd
3
4
1
xs
2
xd
3
xs
4
xd

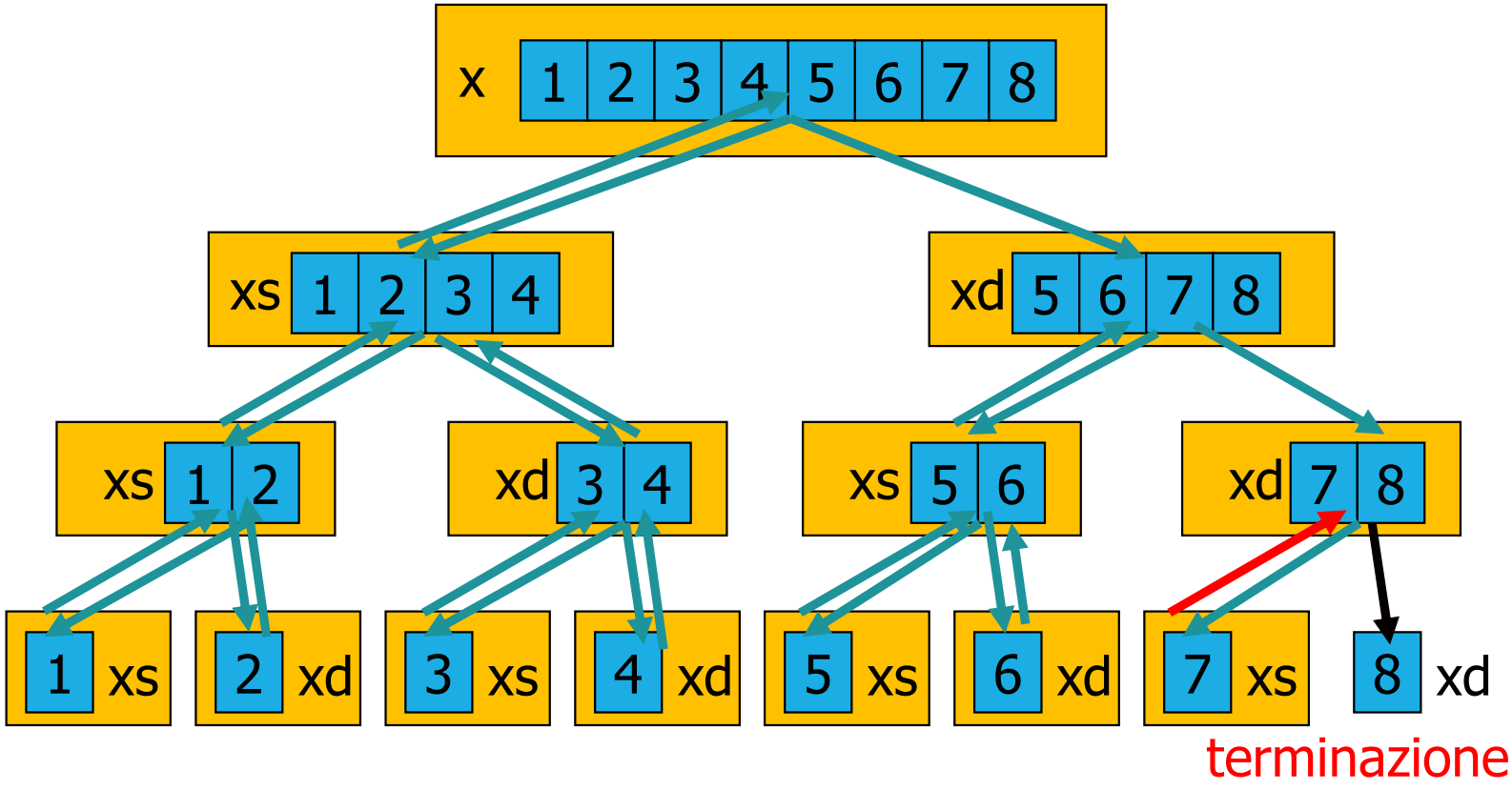

x
1 2 3 4 5 6 7 8
xs
1 2 3 4
xd
5 6 7 8
xs
1 2
xd
3 4
xs
5 6
1 xs
2 xd
3 xs
4 xd

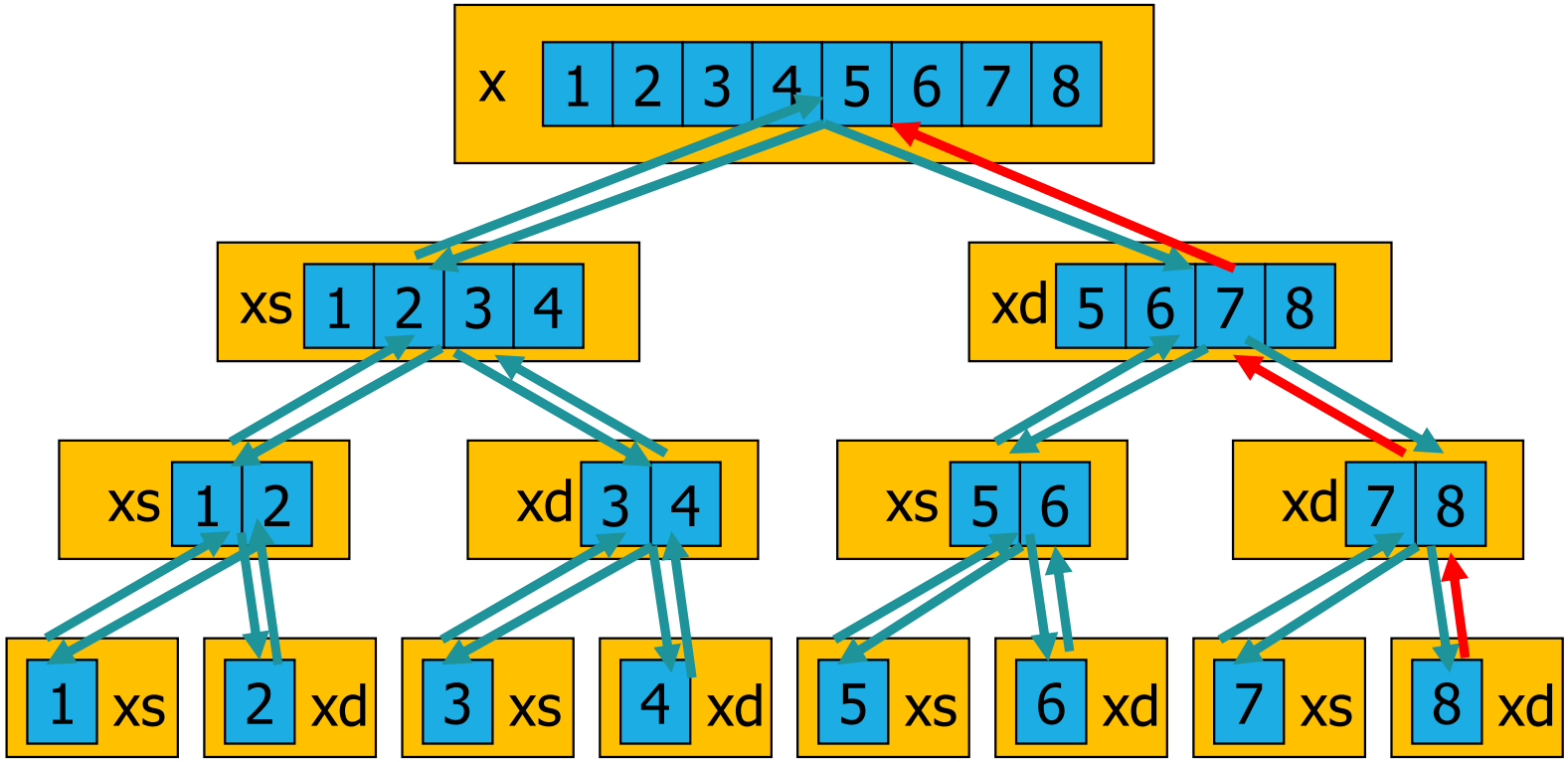

x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
1 xs
2 xd
3 xs
4 xd
5 xs
terminazione


x
1 2 3 4 5 6 7 8
xs
1 2 3 4
xd
5 6 7 8
xs
1 2
xd
3 4
xs
5 6
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd
terminazione


x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
xd 7 8
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd


x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
xd 7 8
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd
7 xs
terminazione


x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
xd 7 8
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd
7 xs
8 xd
terminazione


x 1 2 3 4 5 6 7 8
xs 1 2 3 4
xd 5 6 7 8
xs 1 2
xd 3 4
xs 5 6
xd 7 8
1 xs
2 xd
3 xs
4 xd
5 xs
6 xd
7 xs
8 xd

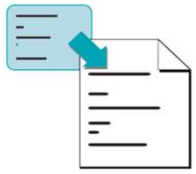
```
void show(int x[], int l, int r) {
  int i, c;
  if (l >= r)
    return;
  c = (r+l)/2;
  printf("xs = ");
  for (i=l; i <= c; i++)
    printf("%d", x[i]);
  printf("\n");
  show(x, l, c);
  printf("xd = ");
  for (i=c+1; i <= r; i++)
    printf("%d", x[i]);
  printf("\n");
  show(x, c+1, r);
  return;
}
```

condizione di terminazione

divisione

chiamata ricorsiva

chiamata ricorsiva



00show_recursion_tree.c

divide and conquer
a = 2 b = 2

Esempio 2:

dato un vettore di $n=2^k$ interi, determinarne il massimo.

Massimo di un vettore di interi:

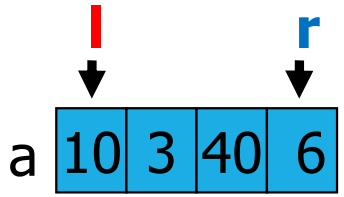
- se la dimensione è n=1, l'unico elemento è anche il massimo
- per n>1
  - dividere il vettore in due sottovettori metà
  - applicare ricorsivamente la ricerca del massimo a ciascun sottovettore
  - confrontare i risultati e restituire il più grande.

Nel main:

```
result = max(a, 0, 3);
```

$n = 2^2$

l = 0 r = 3

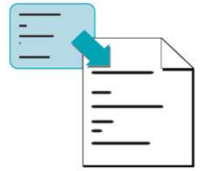



```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

Tracciamento della ricorsione

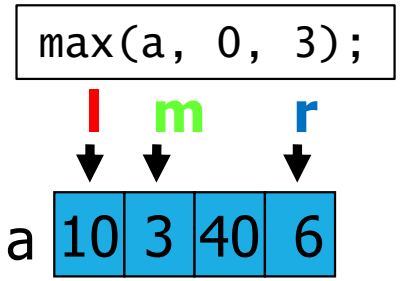
ridondante, solo per chiarezza.
Alternativa:

```
if (u > v)
  return u;
return v;
```



01max_array.c

l = 0 r = 3 m = 1




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```

a | 10 | 3 | 40 | 6 |

l = 0 **r** = 3 **m** = 1

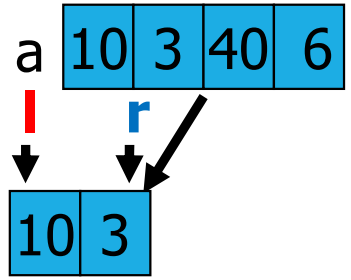
chiamata ricorsiva

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```

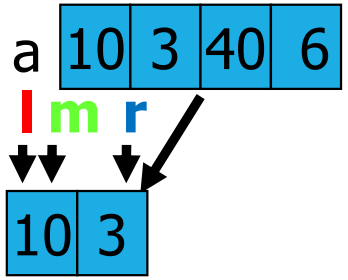
l = 0 r = 1




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```

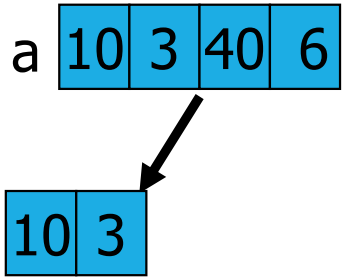
l = 0 r = 1 m = 0




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```

l = 0 r = 1 m = 0



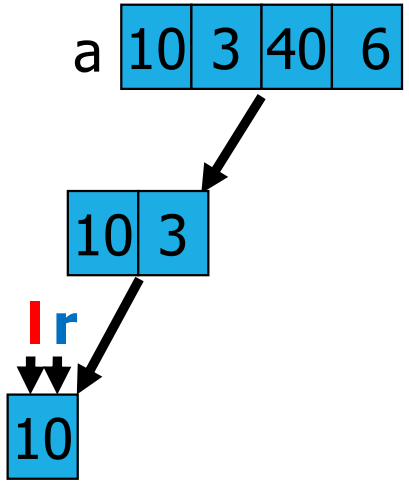

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 0, 0);
```

```
max(a, 0, 0);
```

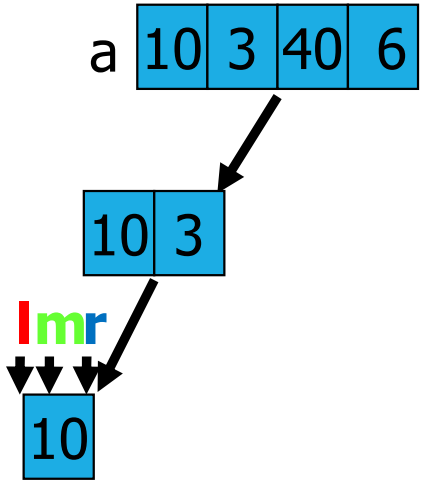
l = 0 r = 0




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 0);
```

l = 0 r = 0 m = 0

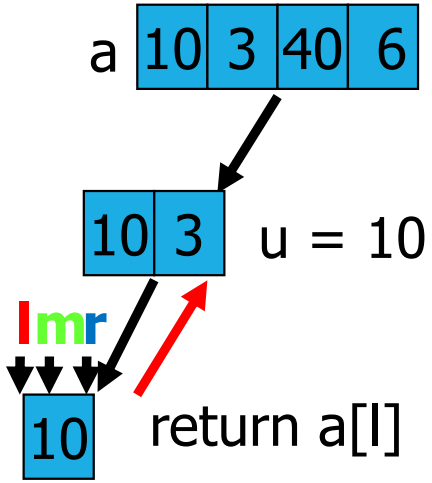



```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

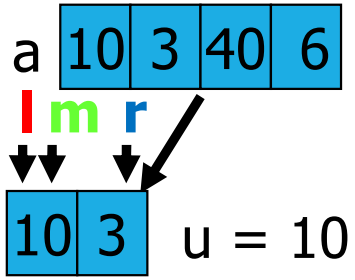
```
max(a, 0, 0);
```

l = 0 r = 0 m = 0



```
max(a, 0, 1);
```

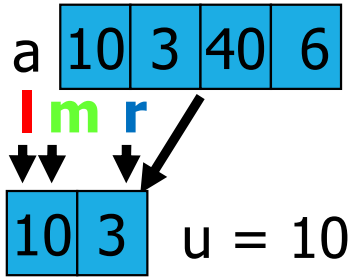
l = 0 r = 1 m = 0




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 0, 1);
```

l = 0 r = 1 m = 0



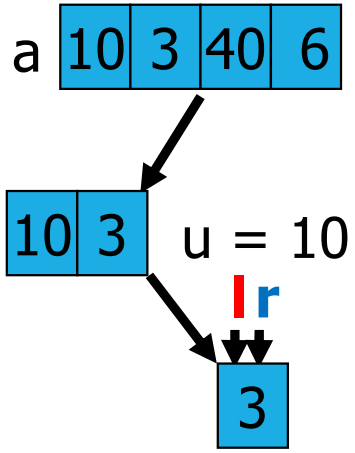

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 1, 1);
```

```
max(a, 1, 1);
```

l = 1 r = 1

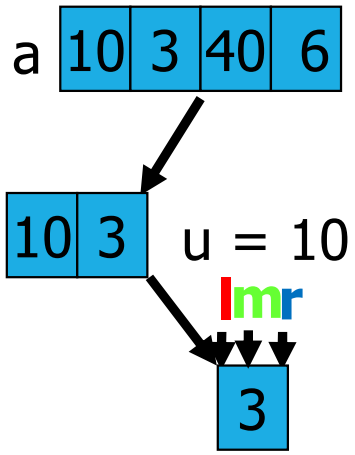



```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

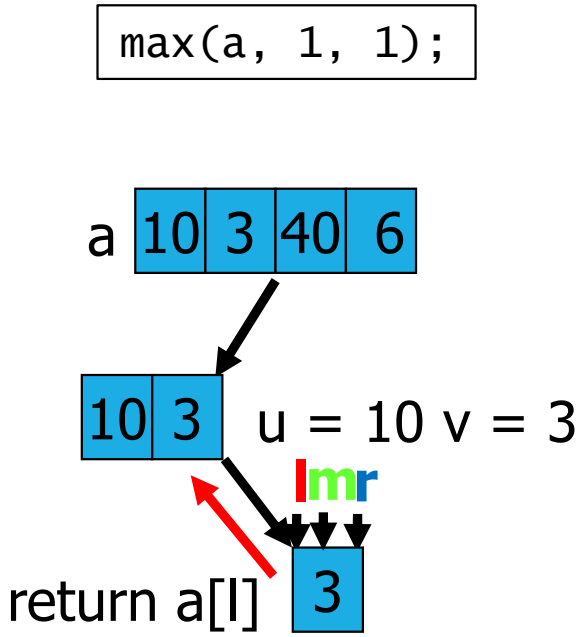
l = 1 r = 1 m = 1

```
max(a, 1, 1);
```



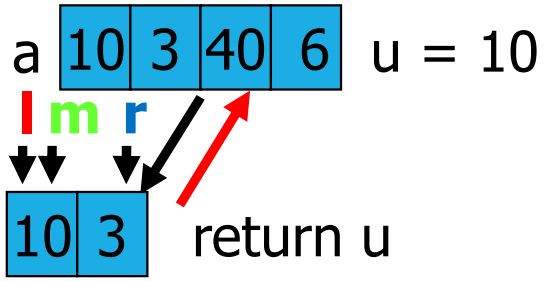
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

l = 1 r = 1 m = 1



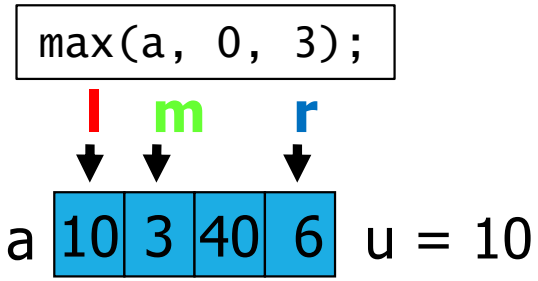
```
max(a, 0, 1);
```

l = 0 r = 1 m = 0

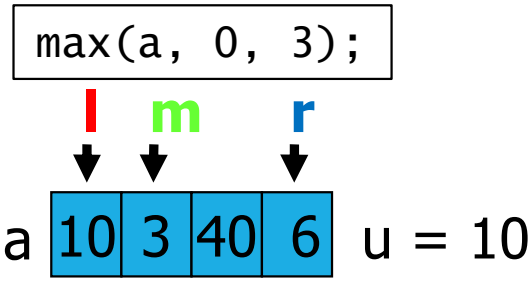



```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

l = 0 r = 3 m = 1




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```



l = 0 r = 3 m = 1

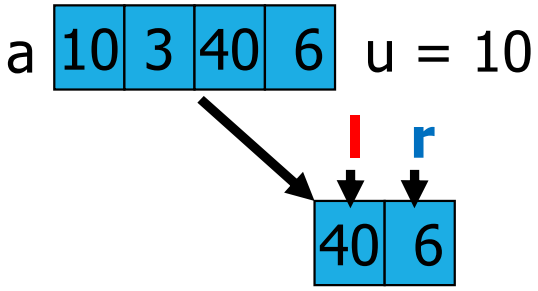
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 2, 3);
```

```
max(a, 2, 3);
```

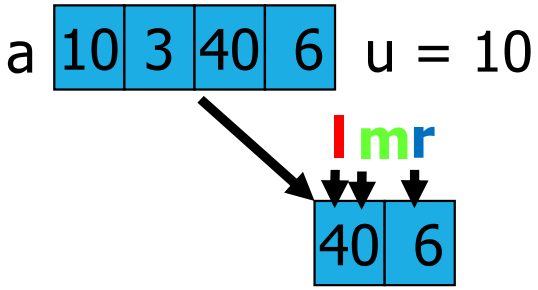
l = 2 r = 3




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 3);
```

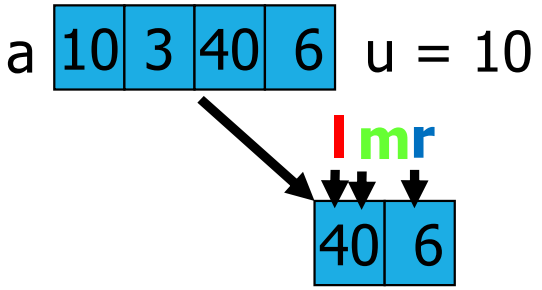
l = 2 r = 3 m = 2




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 3);
```

l = 2 r = 3 m = 2



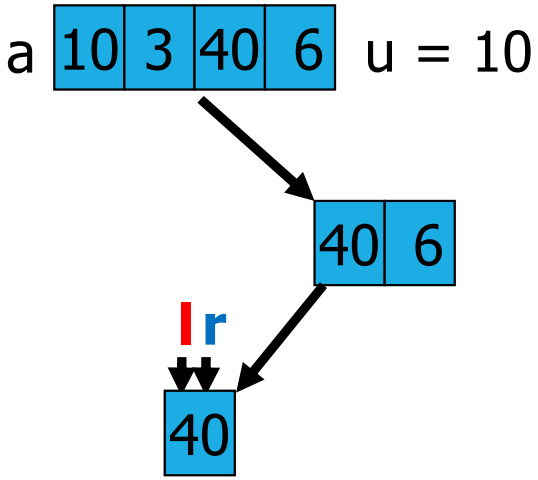

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 2, 2);
```

```
max(a, 2, 2);
```

l = 2 r = 2

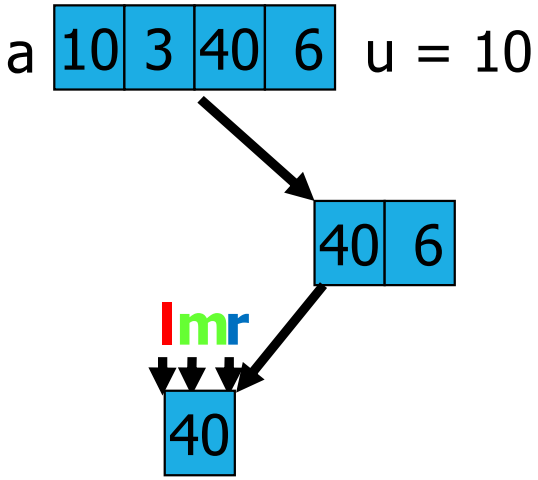



```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

l = 2 r = 2 m = 2

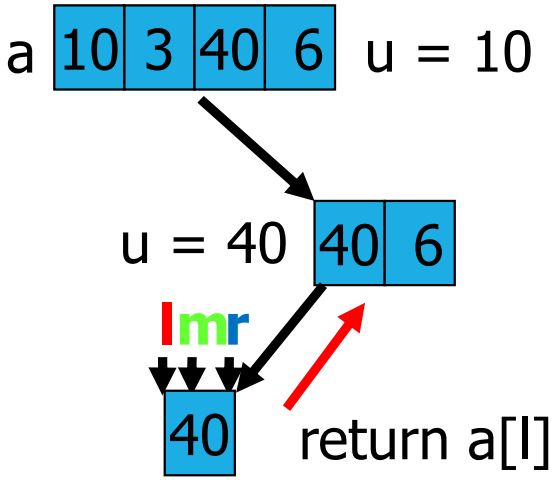
```
max(a, 2, 2);
```



```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

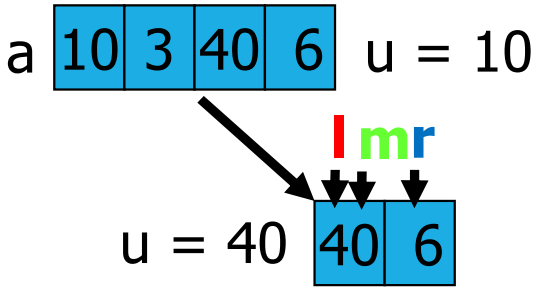
l = 2 r = 2 m = 2

```
max(a, 2, 2);
```



```
max(a, 2, 3);
```

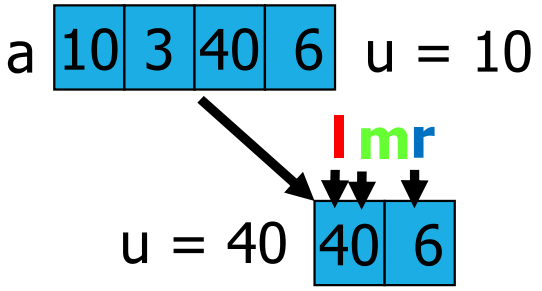
l = 2 r = 3 m = 2




```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
max(a, 2, 3);
```

l = 2 r = 3 m = 2



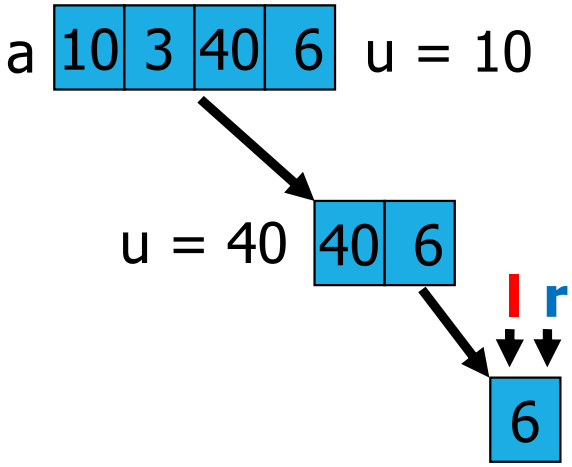

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

chiamata ricorsiva

```
max(a, 3, 3);
```

```
max(a, 3, 3);
```

l = 3 r = 3

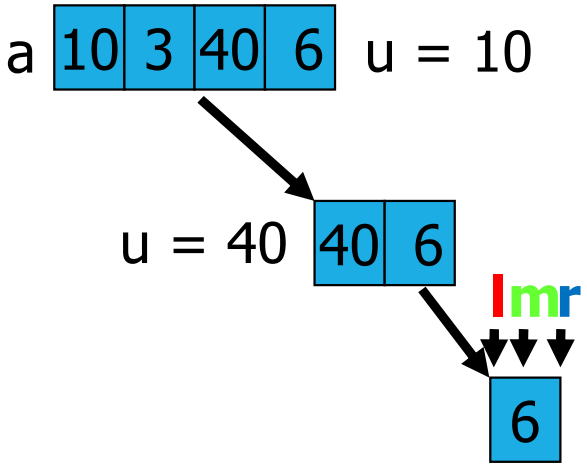



```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

l = 3 r = 3 m = 3

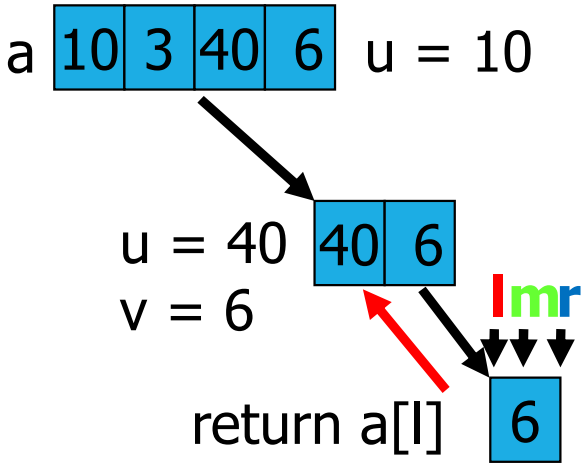
```
max(a, 3, 3);
```



```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

l = 3 r = 3 m = 3

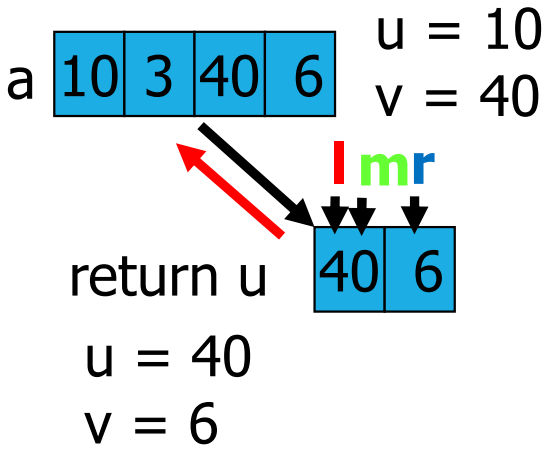
```
max(a, 3, 3);
```



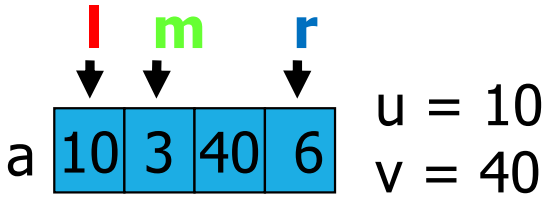
```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

l = 2 r = 3 m = 2

```
max(a, 2, 3);
```



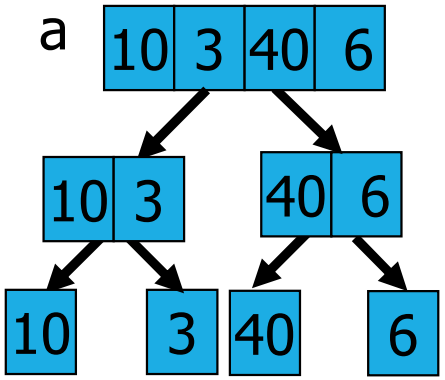
```
result = max (A, 0, 3);
```
⇒ result = 40



l = 0 r = 3 m = 1

```
int max(int a[],int l,int r){
  int u, v;
  int m = (l + r)/2;
  if (l == r)
    return a[l];
  u = max (a, l, m);
  v = max (a, m+1, r);
  if (u > v)
    return u;
  else
    return v;
}
```

return v


a
10
3
40
6
10
3
40
6
10
3
40
6

Analisi di Complessità

Equazione alle Ricorrenze:

T(n) viene espressa in termini di:

- D(n): costo della divisione
- tempo di esecuzione per input più piccoli (ricorsione)
- C(n): costo della ricombinazione

Si suppone che il costo della soluzione elementare sia unitario $\Theta(1)$.

Se:

- a è il numero di sottoproblemi che risulta dalla fase di Divide
- b è il fattore di riduzione, quindi n/b è la dimensione di ciascun sottoproblema

l'equazione alle ricorrenze ha forma:

$$T(n) = D(n) + a\,T(n/b) + C(n) \qquad n > c$$

$$T(n) = \Theta(1) \qquad n \leq c$$

T(n)

Risolvi(Problema):

- Se il problema è elementare:
  - Return Soluzione = Risolvi_banale(Problema)
- Altrimenti:
  - $\text{Sottoproblema}_{1,2,3,\ldots,a}$ = Dividi(Problema) ;
  - Per ciascun Sottoproblema_i:
    - Sottosoluzione_i = Risolvi(Sottoproblema_i) ;
  - Return Soluzione = Combina($\text{Sottosoluzione}_{1,2,3,\ldots,a}$) ;

Θ(1)

D(n)

a sottoproblemi

T(n/b)

C(n)

Se:

- a è il numero di sottoproblemi che risulta dalla fase di Divide
- la riduzione è di un valore k_i, che può variare di passo in passo

l'equazione alle ricorrenze ha forma:

$T(n) = D(n) + \sum_{i=0}^{a-1} T(n-k_i) + C(n)$ $\quad n > c$

$T(n) = \Theta(1)$ $\quad n \leq c$

T(n)

Risolvi(Problema):

- Se il problema è elementare:
  - Return Soluzione = Risolvi_banale(Problema)
- Altrimenti:
  - $\text{Sottoproblema}_{1,2,3,\ldots,a}$ = Dividi(Problema) ;
  - Per ciascun Sottoproblema_i:
    - Sottosoluzione_i = Risolvi(Sottoproblema_i) ;
  - Return Soluzione = Combina($\text{Sottosoluzione}_{1,2,3,\ldots,a}$) ;

Θ(1)

D(n)

a sottoproblemi

$T(n-k_i)$

C(n)

Problemi ricorsivi semplici

Matematici:

- fattoriale
- numeri di Fibonacci
- massimo comun divisore
- prodotto di 2 interi positivi.

Il fattoriale

decrease and conquer
a = 1 k_i = 1

Fattoriale (definizione ricorsiva)

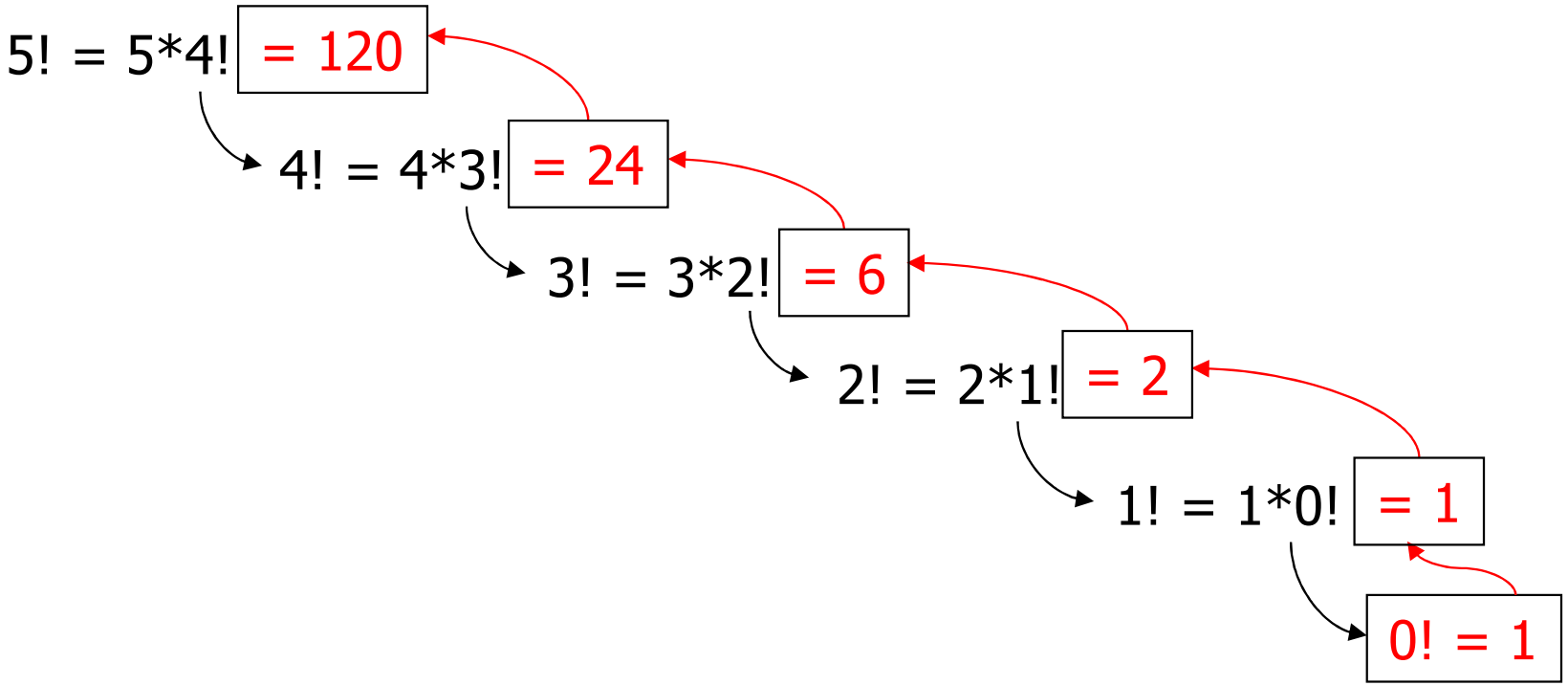
$n! \equiv n * (n-1)!$ $\quad n \geq 1$

$0! \equiv 1! \equiv 1$

Fattoriale (definizione iterativa)

$n! \equiv \prod_{i=0}^{n-1}(n-i) = n * (n-1) * \ldots. 2 * 1$

Esempio



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

02recursive_factorial.c

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$
- equazione alla ricorrenze:

$$T(n) = \Theta(1) + T(n-1) \qquad n > 1$$
$$T(1) = \Theta(1)$$

Risoluzione per sviluppo (unfolding):

T(n) = 1 + T(n-1)

T(n-1) = 1 + T(n-2)

T(n-2) = 1 + T(n-3)

Sostituendo in T(n)

T(n) = 1+1+1+T(n-3) = $\sum_{i=0}^{n-1} 1 = 1 + (n-1) = n$

terminazione:
n-i = 1
i= n -1

Quindi:

T(n) = O(n)

Esistono anche altre tecniche di risoluzione di equazioni alle ricorrenze che permettono di esprimere limiti stretti di tipo Θ anziché O.

I numeri di Fibonacci

Numeri di Fibonacci:

$FIB_n = FIB_{n-2} + FIB_{n-1}$ $\quad n>1$

$FIB_0 = 0$

$FIB_1 = 1$

decrease and conquer
$a = 2$ $k_i = 1$ $k_{i-1} = 2$

Leonardo Pisano (Fibonacci)

- Leonardo detto “Bigollo” Pisano (figlio di Bonaccio $\Rightarrow$ Fibonacci) (circa 1170-1240)
- Matematico pisano che introdusse in Europa la numerazione indo-arabica nel Liber abaci (1202)
- Quesito: «*quante coppie di conigli nasceranno in un anno, a partire da un’unica coppia, se ogni mese ciascuna coppia dà alla luce una nuova coppia che diventa produttiva a partire dal secondo mese?*». Si ipotizza che nessun coniglio muoia.

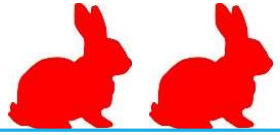
Mese 0

$FIB_0 = 0$

rosso: coppia non fertile **verde**: coppia fertile

⬇ diventa fertile ⬇ sopravvive ↘ genera

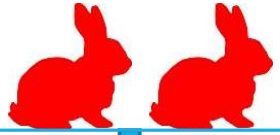
Mese 0 $FIB_0 = 0$

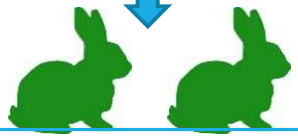
Mese 1  $FIB_1 = 1$

rosso: coppia non fertile **verde**: coppia fertile

⬇ diventa fertile ⬇ sopravvive ↘ genera

Mese 0 $FIB_0 = 0$

Mese 1  $FIB_1 = 1$

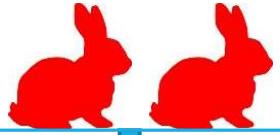
Mese 2  $FIB_2 = 1$

rosso: coppia non fertile **verde**: coppia fertile

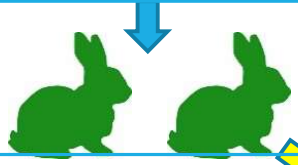
diventa fertile sopravvive genera

Mese 0 $FIB_0 = 0$

Mese 1 $FIB_1 = 1$



Mese 2 $FIB_2 = 1$



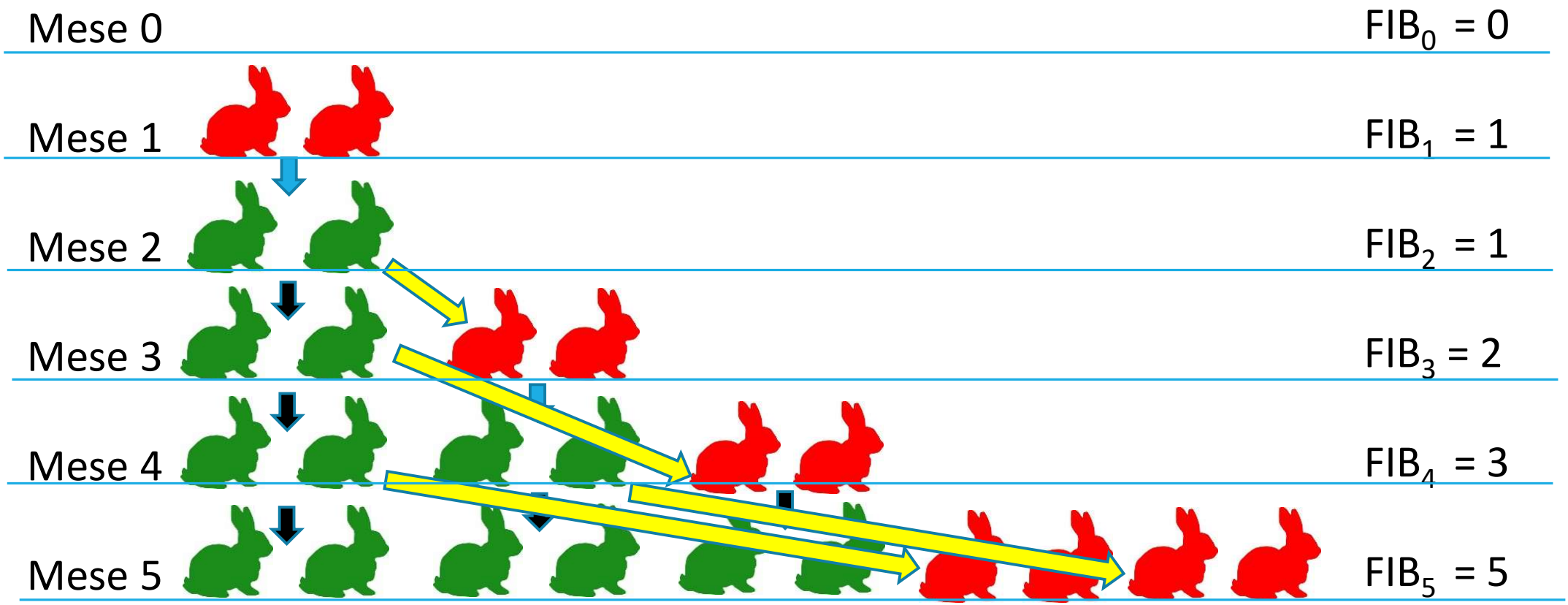
Mese 3 $FIB_3 = 2$

rosso: coppia non fertile **verde**: coppia fertile

diventa fertile sopravvive genera

rosso: coppia non fertile **verde**: coppia fertile

diventa fertile sopravvive genera



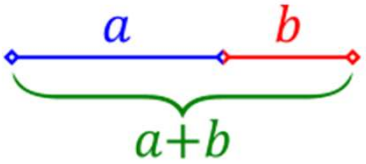
rosso: coppia non fertile **verde**: coppia fertile

diventa fertile sopravvive genera

La sezione aurea φ =a/b

La sezione aurea o rapporto aureo o numero aureo o costante di Fidia o proporzione divina è il rapporto fra due segmenti diversi a e b tale per cui il maggiore (a) è **medio proporzionale** tra il minore (b) e la somma dei due. Lo stesso rapporto esiste anche tra il minore e la differenza.

$$(a+b) : a = a : b = b : (a-b)$$



Ponendo $\varphi = a/b$, $(a+b)/a = a/b$ diventa:

$$(\varphi + 1)/\varphi = \varphi \text{ quindi } \varphi^2 - \varphi - 1 = 0$$

Risolvendo e considerando solo la soluzione positiva

$$\varphi = (1 + \sqrt{5})/2 = 1.61803$$

La sezione aurea e i numeri di Fibonacci

Considerando anche la soluzione negativa:

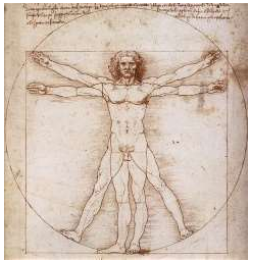
$$\varphi' = (1 - \sqrt{5})/2 = -0.61803$$

l'n-esimo numero di Fibonacci FIB_n si può esprimere come:

$$FIB_n = (\varphi^n - \varphi'^n)/ \sqrt{5}$$

La sezione aurea nella storia

- fra Luca Pacioli (1509) “De divina proportione”
- l’uomo vitruviano (1490) di Leonardo da Vinci
- Modulor (1948) di Le Corbusier
- Lateralus (2001) dei Tool



Lateralus (Tool, 2001)

Traccia 9: Keenan usa le sillabe per formare i primi sei numeri di Fibonacci (partendo da 1):

[1] black

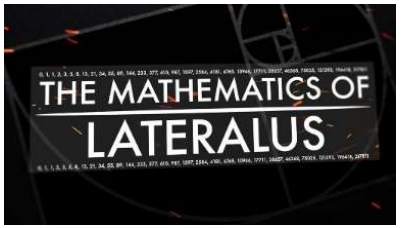
[1] then

[2] white are

[3] all I see

[5] in my infancy

[8] red and yellow then came to be

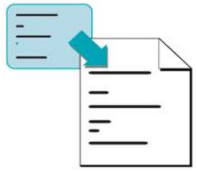


https://youtu.be/uOHkeH2VaE0

PS: la numerologia secondo Umberto Eco
https://www.cicap.org/n/articolo.php?id=273203

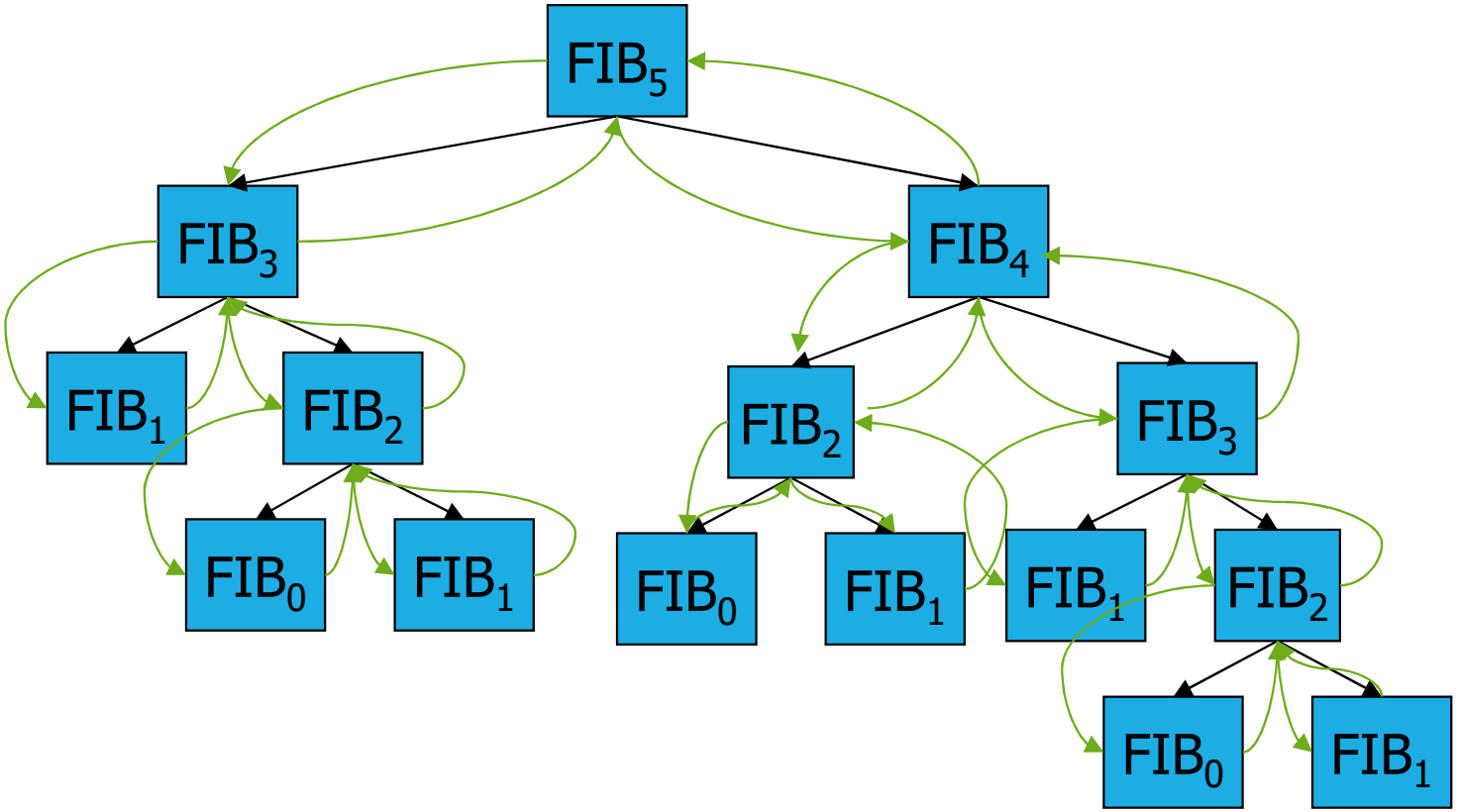
Ricorsione e numeri di Fibonacci

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



03recursive_fibonacci.c

Esempio


FIB5
FIB3
FIB4
FIB1
FIB2
FIB2
FIB3
FIB0
FIB1
FIB0
FIB1
FIB1
FIB2
FIB0
FIB1

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 2$, $k_i = 1$, $k_{i-1} = 2$
- equazione alla ricorrenze:

$$T(n) = 1 + T(n-1) + T(n-2) \qquad n > 1$$

$$T(0) = 1 \quad T(1) = 1$$

- approssimazione conservativa: essendo

$$T(n-2) \leq T(n-1)$$

lo sostituisco con T(n-1) e l'equazione diventa

$$T(n) = 1 + 2T(n-1) \qquad n > 1$$

$$T(0)=T(1) = 1$$

Risoluzione per sviluppo (unfolding):

T(n) = 1 + 2T(n-1)

T(n-1) = 1 + 2T(n-2)

T(n-2) = 1 + 2T(n-3)

Sostituendo in T(n)

T(n) = 1 + 2 + 4 + 2^3T(n-3) = $\sum_{i=0}^{n-1} 2^i = 2^n - 1$

terminazione:
n-i = 1
i= n -1

Quindi:

T(n) = $O(2^n)$

$\sum_{i=0}^{k} x^i = (x^{k+1} - 1)/(x-1)$

Stima migliore: T(n) = $O(\varphi^n)$.

Il massimo comun divisore

decrease and conquer
a = 1 k_i variabile

Il massimo comun divisore *gcd* di due interi *x* e *y* non entrambi nulli è il più grande dei divisori comuni di *x* e *y.* Si assuma che inizialmente *x > y.*

Esempio: gcd(600,54) = 6

Algoritmo inefficiente basato sulla scomposizione in fattori primi:

x = $p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_r^{e_r}$ y = $p_1^{f_1} \cdot p_2^{f_2} \cdot \ldots \cdot p_r^{f_r}$

gcd(x,y) = $p_1^{\min(e_1,f_1)} \cdot p_2^{min(e_2,f_2)} \ldots \cdot p_r^{\min(e_r,fr)}$

Algoritmo di Euclide (IV sec. aC):

versione 1: sottrazione

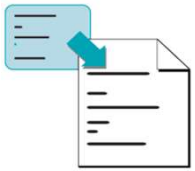
- se x > y

  gcd(x, y) = gcd(x-y, y)
- altrimenti

  gcd(x, y) = gcd(x, y-x)
- terminazione:

  se x=y ritorna x

```
int gcd(int x, int y) {
  if(x == y)
    return x;
  if (x > y)
    return gcd(x-y, y);
  return gcd(x, y-x);
}
```



04recursive_gcd.c

Algoritmo di Euclide-Lamé (1844)-Dijkstra:

versione 2: resto della divisione intera

- gcd(x, y) = gcd(y, x%y)
- terminazione:

  se y=0 ritorna x

```
int gcd(int x, int y) {
  if(y == 0)
     return x;
  return gcd(y, x % y);
}
```

else ridondante eliminato

Esempi:

gcd(600,54)
gcd(54, 6)
gcd(6, 0) return 6

gcd(314159,271828)
gcd(271828,42331)
gcd(42331,17842)
gcd(17842,6647)
gcd(6647,4548)
gcd(4548,2099)
gcd(2099,350)
gcd(350,349)
gcd(349,1)
gcd(1,0) return 1
314159 e 271828 sono coprimi tra di loro

Analisi di complessità

- D(x,y) = $\Theta(1)$, C(x,y) = $\Theta(1)$
- a = 1, riduzione variabile
- Caso peggiore: x e y sono 2 numeri di Fibonacci consecutivi (il loro gcd è 1):

x = FIB(n+1) y = FIB(n)

Equazione alle ricorrenze

T(x,y) = T(FIB(n+1), FIB(n))

= 1 + T(FIB(n),FIB(n+1)%FIB(n)) T(x,0) = 1

ma FIB(n+1)%FIB(n) = FIB(n-1)

terminazione:
n passi

T(x,y) = T(FIB(n+1), FIB(n))

= 1 + T(FIB(n), FIB(n-1))

= $\sum_{i=0}^{n-1} 1 = n$

T(x, y) = O(n), ma, visto che

y = FIB(n) = $(\varphi^n - \varphi'^n)/\sqrt{5} = \Theta(\varphi^n)$, allora

n è una funzione di $\log_\varphi(y)$

Quindi:

T(n) = O(log(y)) e T(x, y) = O(log(y))

Il massimo di un vettore

Analisi di complessità:

- D(n) = $\Theta(1)$, C(n) = $\Theta(1)$
- a = 2, b = 2

divide and conquer
a = 2 b = 2

Equazione alle ricorrenze

T(n) =2T(n/2) + 1 n > 1

T(1) = 1

Risoluzione per sviluppo (unfolding):

T(n) = 1 + 2T(n/2)

T(n/2) = 1 + 2T(n/4)

T(n/4) = 1 + 2T(n/8)

Sostituendo in T(n)

T(n) = 1 + 2 + 4 + 2^3T(n/8)

= $\sum_{i=0}^{log_2 n} 2^i$ = ($2^{\log_2 n + 1}$-1)/(2-1)

= 2 * $2^{\log_2 n}$ -1 = 2n-1

Quindi:

T(n) = O(n)

terminazione:
$n/2^i = 1$
$i = \log_2 n$

$\sum_{i=0}^{k} x^i$ = (x^{k+1} -1)/(x-1)

Il prodotto di 2 interi

Moltiplicazione di 2 interi positivi x e y:

- Algoritmo elementare: moltiplicazione di ogni cifra del moltiplicatore per le cifre del moltiplicando, scalamento a sinistra (moltiplicazione per 10) e somme. Richiede la conoscenza delle tabelline pitagoriche
- Algoritmo ricorsivo elementare
  - se y = 1 allora x * 1 = x
  - altrimenti x * y = x + x * (y – 1)

divide and conquer
a = 4 b = 2

Algoritmo ricorsivo per la moltiplicazione di 2 interi positivi x e y di n cifre (con $n = 2^{k)}$:

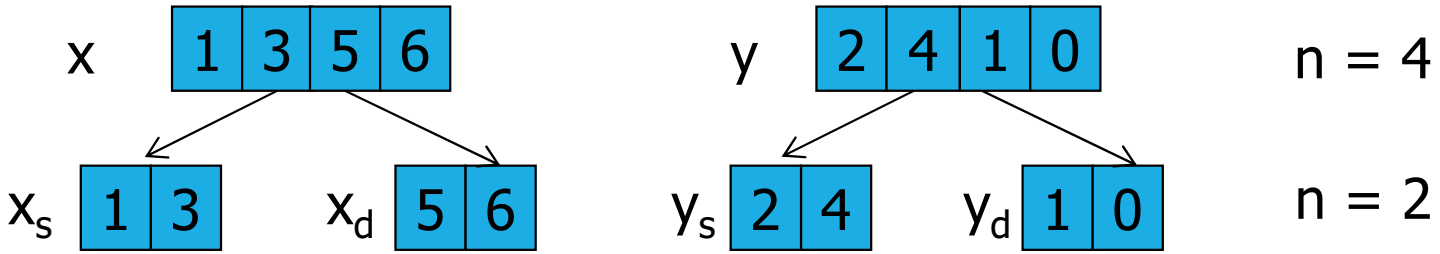
- se la dimensione è n=1, calcola x * y (tabelline pitagoriche)
- per n>1
  - dividi x in 2: $x = 10^{n/2} * x_s + x_d$
  - dividi y in 2: $y = 10^{n/2} * y_s + y_d$
  - calcola ricorsivamente $x_s * y_s$, $x_s * y_d$, $x_d * y_s$, $x_d * y_d$,
  - calcola $x * y = 10^n * x_s * y_s + 10^{n/2} * (x_s * y_d + x_d * y_s) + x_d * y_d$

Esempio

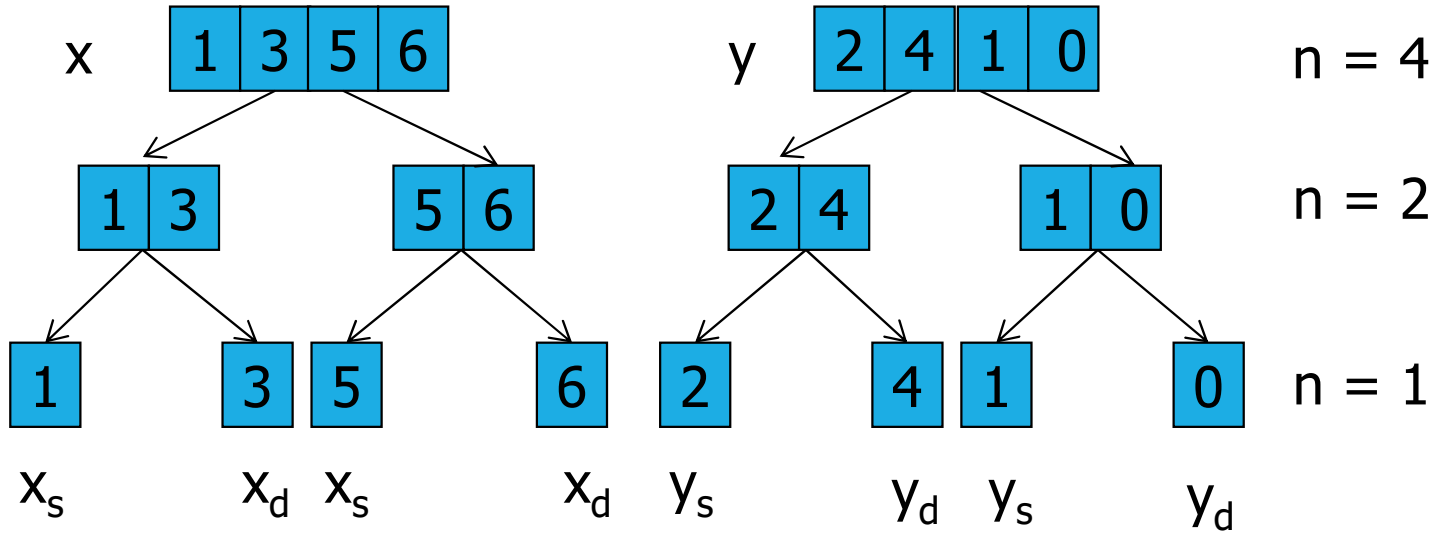
1356 * 2410 = 3.267.960

x | 1 | 3 | 5 | 6 | * y | 2 | 4 | 1 | 0 | n = 4

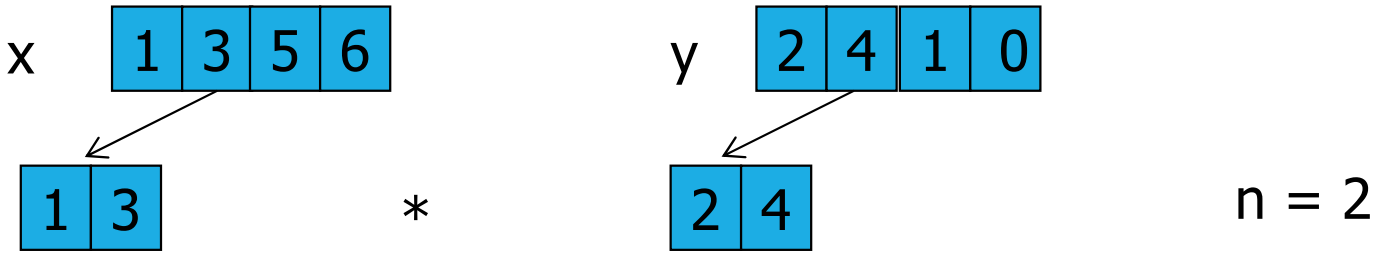
1356 * 2410 = 3.267.960



1356 * 2410 = 3.267.960



$$1356 * 2410 = 3.267.960$$

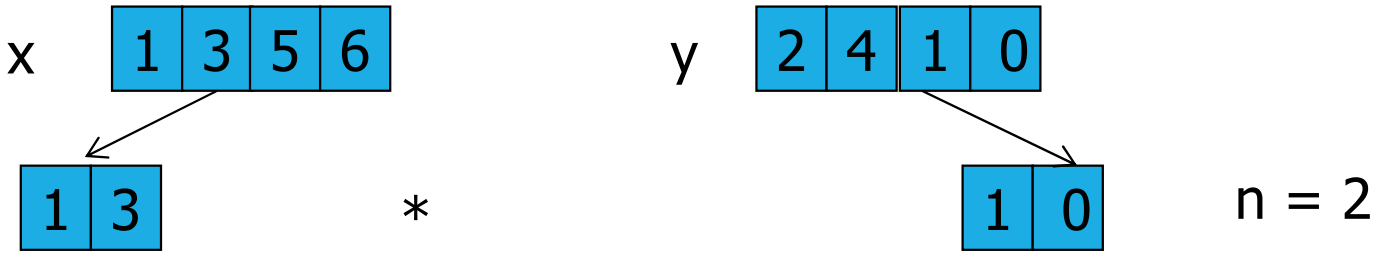


$$10^n \quad x_s \quad y_s \quad 10^{n/2} \quad x_s \quad y_d \quad x_d \quad y_s \quad x_d \quad y_d$$

$$10^2 * 1 * 2 + 10^1 * (1 * 4 + 3 * 2) + 3 * 4$$

$$13 * 24 = 10^2 * 2 + 10^1 * 10 + 12 = 312$$

$$1356 * 2410 = 3.267.960$$

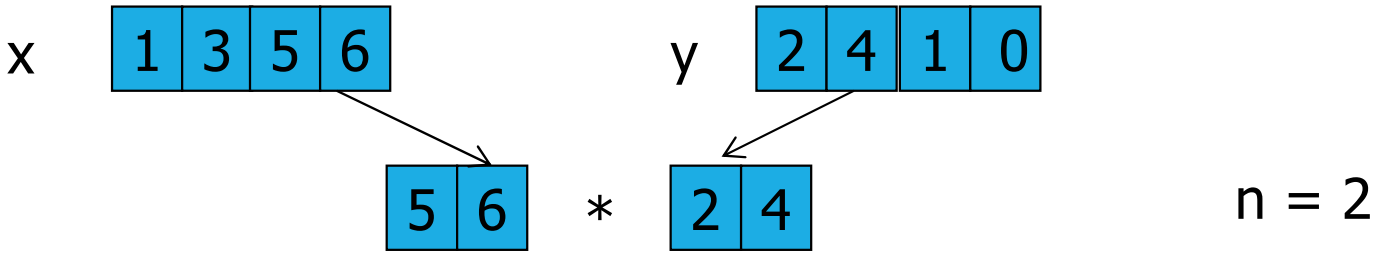


$$10^n \quad x_s \quad y_s \quad 10^{n/2} \quad x_s \quad y_d \quad x_d \quad y_s \quad x_d \quad y_d$$

$$10^2 * 1 * 1 + 10^1 * (1 * 0 + 3 * 1) + 3 * 0$$

$$13 * 10 = 10^2 * 1 + 10^1 * 3 + 0 = 130$$

$$1356 * 2410 = 3.267.960$$

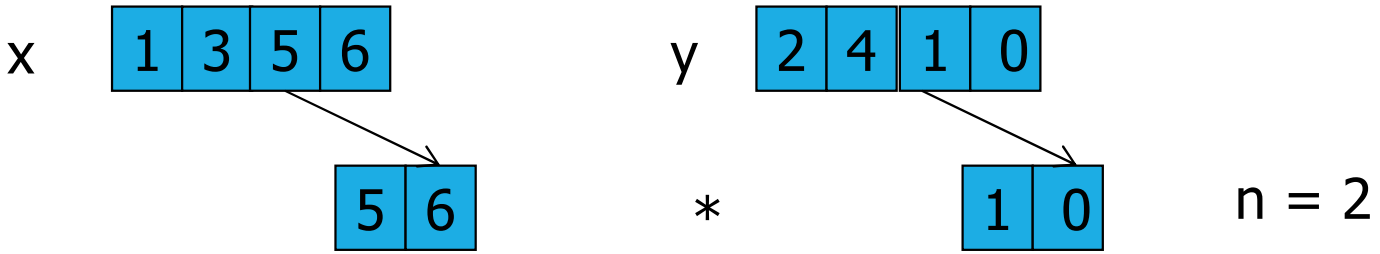


$$10^n \quad x_s \quad y_s \quad 10^{n/2} \quad x_s \quad y_d \quad x_d \quad y_s \quad x_d \quad y_d$$

$$10^2 * 5 * 2 + 10^1 * (5 * 4 + 6 * 2) + 6 * 4$$

$$56 * 24 = 10^2 * 10 + 10^1 * 32 + 24 = 1344$$

$$1356 * 2410 = 3.267.960$$



$$10^{n} \quad x_s \quad y_s \quad 10^{n/2} \quad x_s \quad y_d \quad x_d \quad y_s \quad x_d \quad y_d$$

$$10^{2} * 5 * 1 + 10^{1} * (5 * 0 + 6 * 1) + 6 * 0$$

$$56 * 10 = 10^2 * 5 + 10^1 * 6 + 0 = 560$$

$$1356 * 2410 = 3.267.960$$

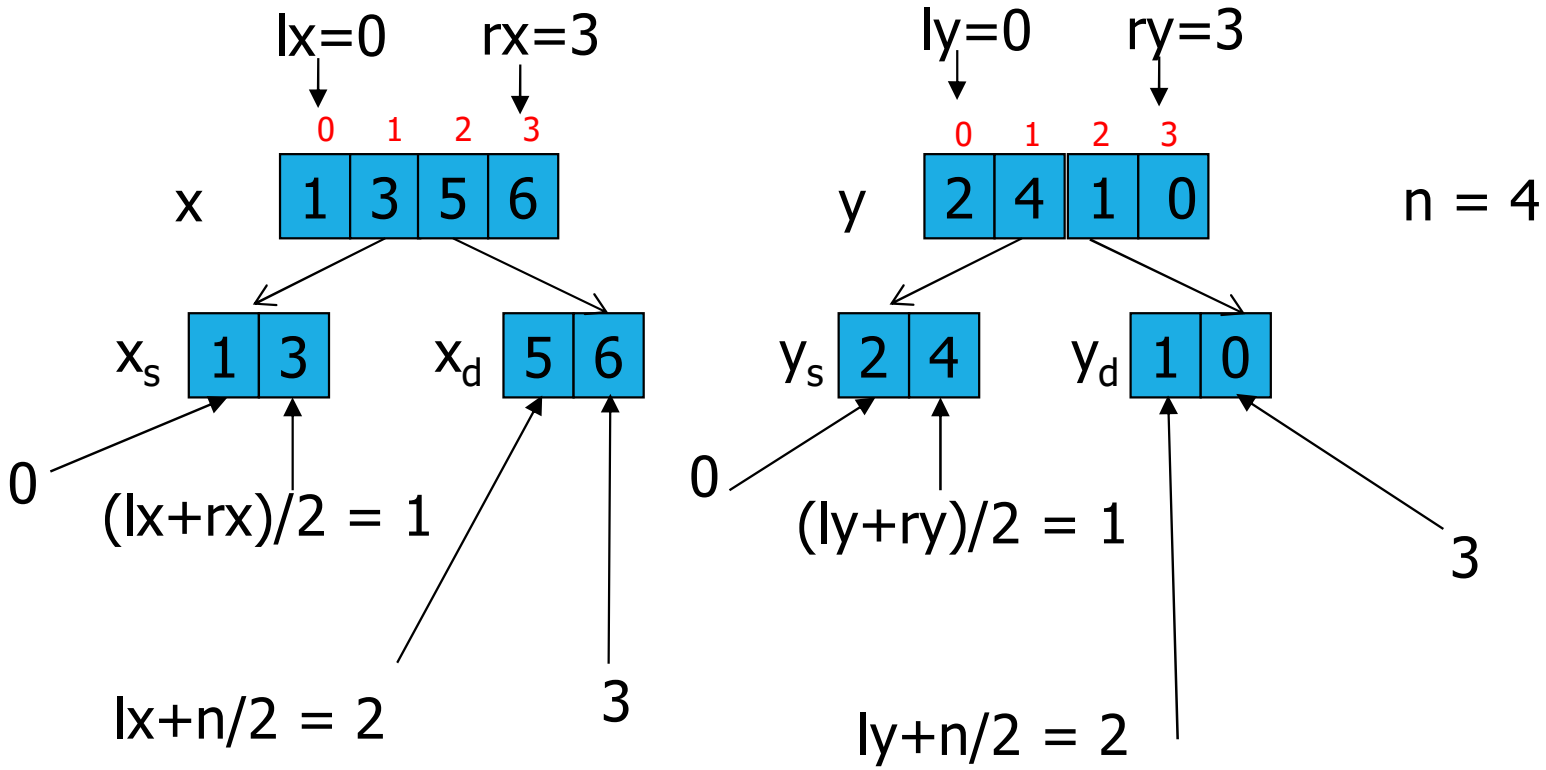
x | 1 | 3 | 5 | 6 | * y | 2 | 4 | 1 | 0 | n = 4

$$10^n \quad x_s \quad y_s \quad 10^{n/2} \quad x_s \quad y_d \quad x_d \quad y_s \quad x_d \quad y_d$$

$$10^4 * 13 * 24 + 10^2 * (13 * 10 + 56 * 24) + 56 * 10$$

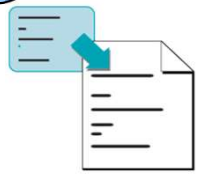
$$1356 * 2410 = 10^4 * 312 + 10^2 * (130 + 1344) + 560 = 3.267.960$$

Identificazione dei sottovettori sinistro e destro:



```
long prod(int *x,int lx,int rx,int *y,int ly,int ry,int n) {
  long t1, t2, t3;
  if (n == 1)
    return (x[lx]*y[ly]);
  t1 = prod(x, lx, (lx+rx)/2, y, ly, (ly+ry)/2, n/2);
  t2 = prod(x, lx, (lx+rx)/2, y, ly+n/2, ry, n/2)
     + prod(x, lx+n/2, rx, y, ly, (ly+ry)/2, n/2);
  t3 = prod(x, lx+n/2, rx, y, ly + n/2, ry, n/2);
  return t1 * pow(10,n) + t2 * pow (10, n/2) + t3;
}
```

operazione considerata elementare



05recursive_integer_product.c

Analisi di complessità

- moltiplicazione per 10^k: shift a sinistra (ogni shift è $\Theta(1)$, in totale è $\Theta(k)$. Al più k=n e sono necessarie 2 moltiplicazioni, il costo è quindi $\Theta(n)$
- la somma di numeri su k cifre è $\Theta(k)$. Due numeri di n cifre danno un prodotto su 2n cifre. Al più k=2n e sono necessarie 3 somme, il costo è quindi $\Theta(n)$
- moltiplicazione: costo della ricorsione (4 moltiplicazioni)
- $D(n) = \Theta(1)$, $C(n) = \Theta(n)$, $D(n) + C(n) = \Theta(n)$
- $a = 4$, $b = 2$

Equazione alle ricorrenze:

$T(n) = 4T(n/2) + n \qquad n > 1$

$T(1) = 1$

Risoluzione per sviluppo (unfolding)

$T(n/2) = 4T(n/4) + n/2$

$T(n/4) = 4T(n/8) + n/4$ etc.

$T(n) = n + 4*(n/2) + 4^2 *(n/4) + 4^3 *T(n/8)$

$= \Sigma_{0 \le i \le \log 2n} 4^i / 2^i * n = n * \Sigma_{0 \le i \le \log 2n} 2^i$

$= n*(2^{\log 2n + 1}-1)/(2-1) = n *(2 * 2^{\log 2n} -1) = 2n^2-n$

$T(n) = O(n^2)$

terminazione:
$n/2^i = 1$
$i = \log_2 n$

$\sum_{i=0}^{k} x^i = (x^{k+1} -1)/(x-1)$

divide and conquer
a = 3 b = 2

Algoritmo di Karatsuba e Ofman (1962):

Ottimizzazione dell'algoritmo di base mediante riduzione del numero di moltiplicazioni:

$x_s * y_d + x_d * y_s = x_s * y_s + x_d * y_d - (x_s - x_d) * (y_s - y_d)$

3 moltiplicazioni ricorsive, anziché 4.

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(n)$
- $D(n) + C(n) = \Theta(n)$
- $a = 3$, $b = 2$

Equazione alle ricorrenze:

$T(n) = 3T(n/2) + n \qquad n > 1$

$T(1) = 1$

Risoluzione per sviluppo (unfolding)

$T(n/2) = 3T(n/4) + n/2$

$T(n/4) = 3T(n/8) + n/4$ etc.

$T(n) = n + 3*(n/2) + 3^2*(n/4) + 3^3 *T(n/8)$

$= \sum_{0 \le i \le \log_2 n} 3^i / 2^i * n = n * \sum_{0 \le i \le \log_2 n} (3/2)^i$

$= n*((3/2)^{\log 2n + 1} - 1)/(3/2-1)$

$= 2n * (3/2 *(3^{\log 2n} / 2^{\log 2n}) - 1)$

$= 2n*(3*n^{\log_2 3} /2n-1)$

$= 3n^{\log_2 3} - 2n$

$T(n) = O(n^{\log_2 3})$

terminazione:
$n/2^i = 1$
$i = \log_2 n$

$\sum_{i=0}^{k} x^i = (x^{k+1} - 1)/(x-1)$

$a^{\log_b n} = n^{\log_b a}$

Problemi ricorsivi semplici

Informatici:

- ricerca binaria o dicotomica
- stampa in ordine inverso
- elaborazione di liste concatenate
  - conteggi
  - attraversamenti
  - cancellazione
- alberi binari
  - calcolo di parametri
  - visite
  - espressioni

La ricerca binaria

divide and conquer
a = 1 b = 2

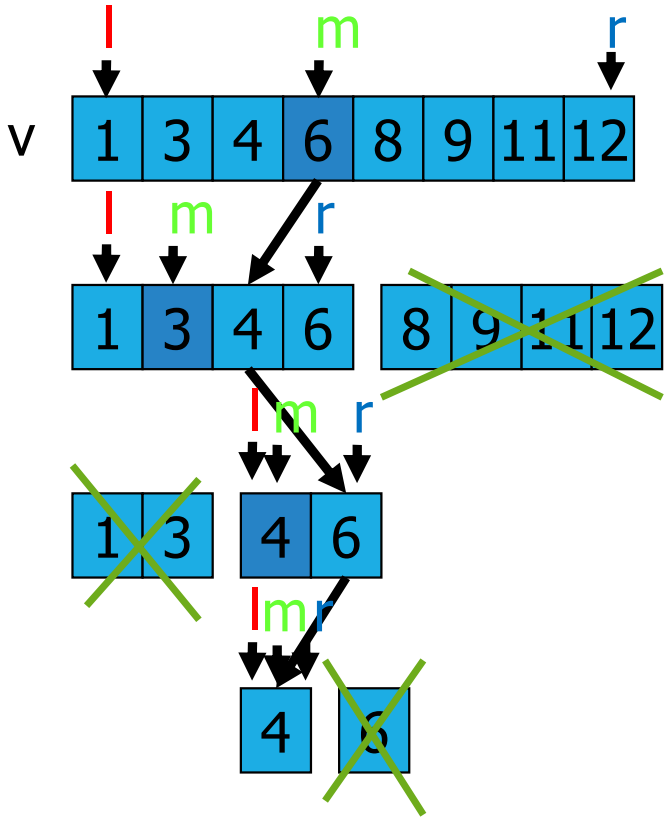
Ricerca binaria o dicotomica: la chiave k è presente all'interno di un vettore ordinato v[n]? Sì/No

Approccio

ipotesi: $n = 2^p$

- ad ogni passo: confronto k con elemento centrale del vettore:
  - =: terminazione con successo
  - <: la ricerca prosegue nel sottovettore di SX
  - >: la ricerca prosegue nel sottovettore di DX
- Terminazione: il vettore non è più significativo (l'indice di sinistra ha scavalcato quello di destra)

Esempio



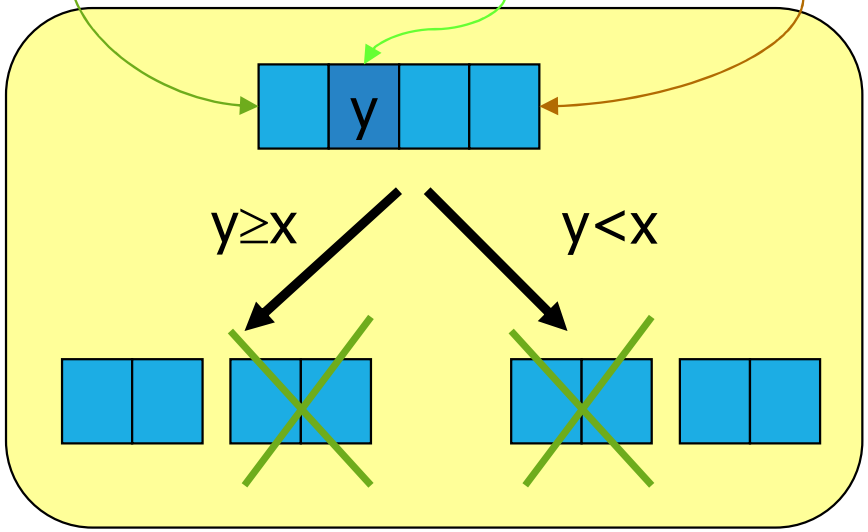
k 4

y = elemento di mezzo

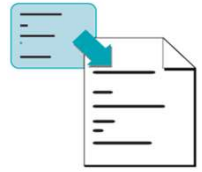
l = indice estremo di SX

r = indice estremo di DX

m = indice elemento di mezzo



```
int BinSearch(int v[], int l, int r, int k) {
  int m;
  if (l > r)
    return -1;

  m = (l + r)/2;
  if (k == v[m])
    return (m);
  if (k < v[m])
    return BinSearch(v, l, m-1, k);

  return BinSearch(v, m+1, r, k);
}
```



07recursive_binsearch.c

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1, b = 2$

Equazione alla ricorrenze:

$T(n) = T(n/2) + 1 \qquad n > 1$

$T(1) = 1$

Risoluzione per sviluppo (unfolding)

$T(n/2) = T(n/4) + 1$

$T(n/4) = T(n/8) + 1$

$T(n) = 1 + 1 + 1 + T(n/8)$

$= \sum_{i=0}^{log_2 n} 1$

$= 1 + \log_2 n$

$T(n) = O(\log n)$

terminazione:
$n/2^i = 1$
$i = \log_2 n$

Stampa in ordine inverso

decrease and conquer
$a = 1$ $k_i = 1$

- Leggere da input una stringa
- Stamparla in ordine inverso

```
void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

07reverse_print.c

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 1$, $k_i = 1$

Equazione alla ricorrenze:

$T(n) = 1 + T(n-1)$ $\quad n > 1$

$T(1) = 1$

$T(n) = O(n)$

Problemi ricorsivi semplici

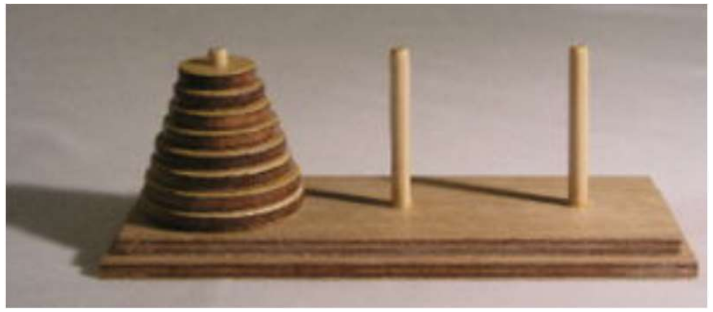
Matematica ricreativa:

- Le Torri di Hanoi
- Il righello.

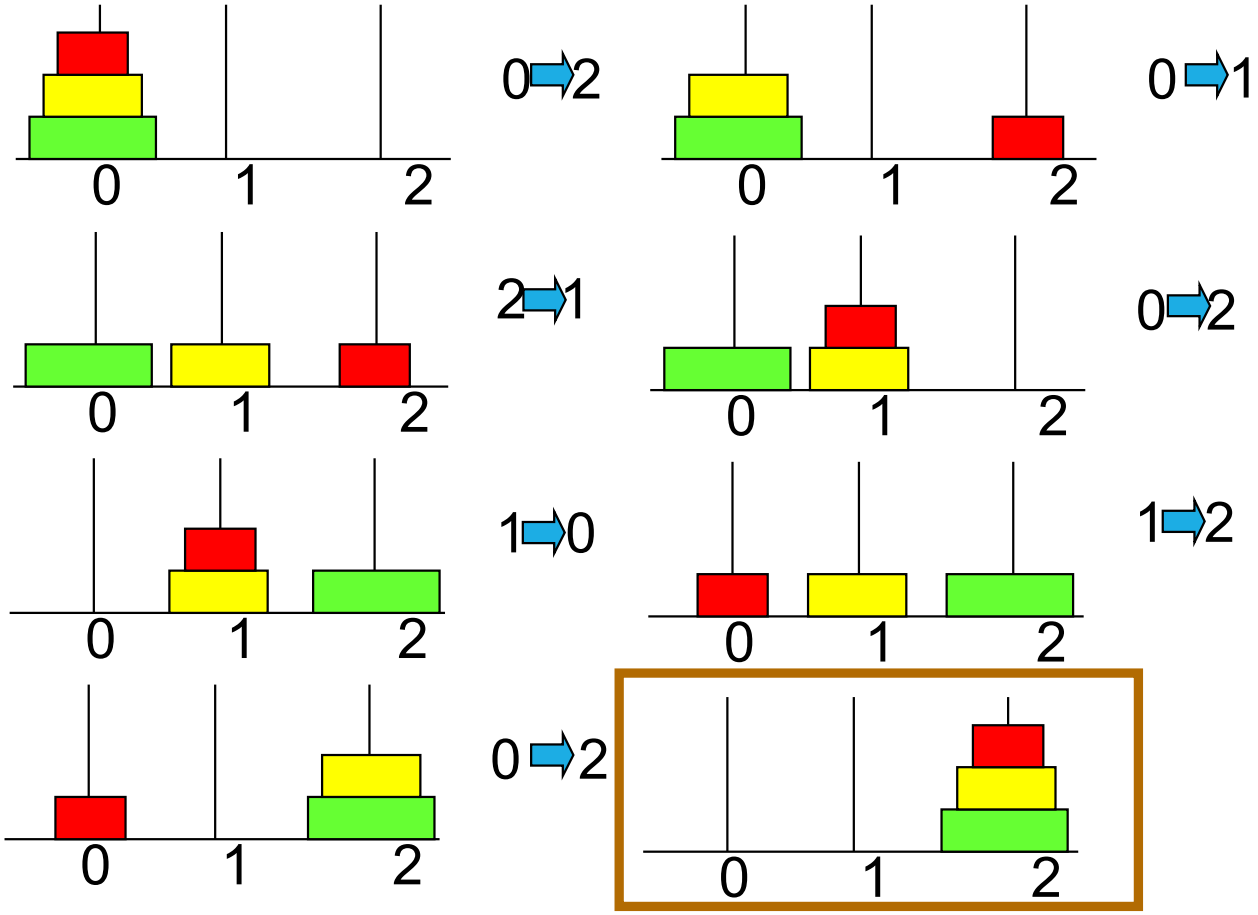
Le Torri di Hanoi (E. Lucas 1883)

decrease and conquer
$a = 2$ $k_i = 1$

- Configurazione iniziale:
  - vi sono 3 pioli, 3 dischi di diametro decrescente sul primo piolo
- Configurazione finale:
  - 3 dischi sul terzo piolo
- Regole:
  - accesso solo al disco in cima
  - sopra ogni disco solo dischi più piccoli
- Generalizzabile a n dischi e k pioli.



Esempio di soluzione



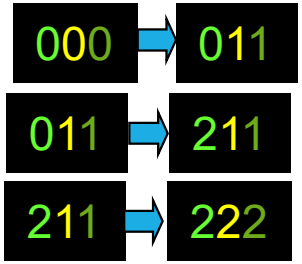
Strategia divide et impera

- Problema iniziale: spostare n dischi da 0 a 2
- Riduzione a sottoproblemi:
  - n-1 dischi da 0 a 1, 2 deposito
  - l'ultimo disco da 0 a 2
  - n-1 dischi da 1 a 2, 0 deposito
- Condizione di terminazione: si muove 1 solo disco.
  - 0, 1, 2: piolo 0, 1, 2
  - disco grande, disco medio, disco piccolo
  - 0 significa disco piccolo su piolo 0, 2 significa disco grande su piolo 2, etc.
  - stato 011
  - transizione di stato ⇨

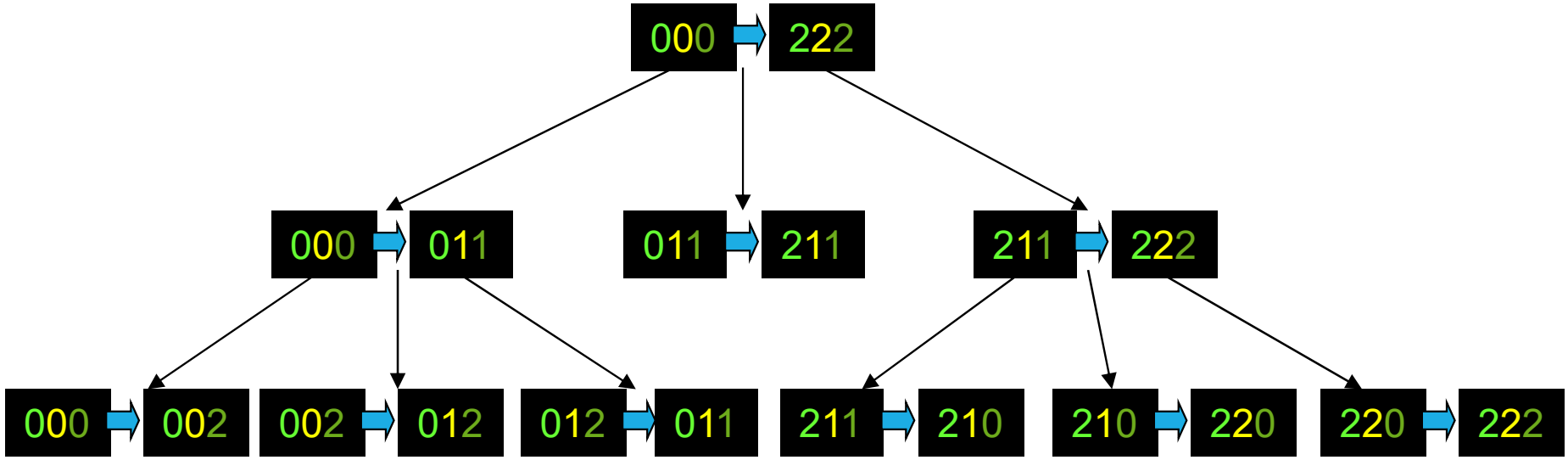
Problema 

decomposto in 3 sottoproblemi di cui 1 elementare:

1. dischi medio e piccolo da 0 a 1
2. disco grande da 0 a 2
3. dischi medio e piccolo da 1 a 2.



Albero della ricorsione



```
void Hanoi(int n, int src, int dest) {
  int aux;
  aux = 3 - (src + dest);
  if (n == 1) {
    printf("src %d -> dest %d \n", src,dest);
    return;
  }
  Hanoi(n-1, src, aux);
  printf("src %d -> dest %d \n", src, dest);
  Hanoi(n-1, aux, dest);
}
```

terminazione

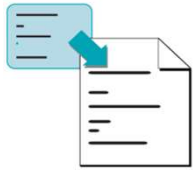
chiamata ricorsiva

divisione

chiamata ricorsiva

soluzione elementare

divisione



8towers_of_hanoi.c

Analisi di complessità

- Dividi: considera n-1 dischi: $D(n)=\Theta(1)$
- Risolvi: risolve 2 sottoproblemi di dimensione n-1 ciascuno: $2T(n-1)$
- Terminazione: spostamento di 1 disco: $\Theta(1)$
- Combina: nessuna azione: $C(n) = \Theta(1)$

Equazione alle ricorrenze:

$$T(n) = 2T(n-1) + 1 \qquad n>1$$
$$T(1) = 1$$

$$T(n) = 1 + 2 + 4 + 8T(n-3)$$
$$= \sum_{0 \le i \le n-1} 2^i$$
$$= 2^{n-1+1} - 1/(2-1)$$
$$= 2^n - 1$$

$T(n) = O(2^n)$

Il righello

divide and conquer
a = 2 b = 2

Tracciare una tacca in ogni punto tra 0 e 2^n estremi esclusi, dove:

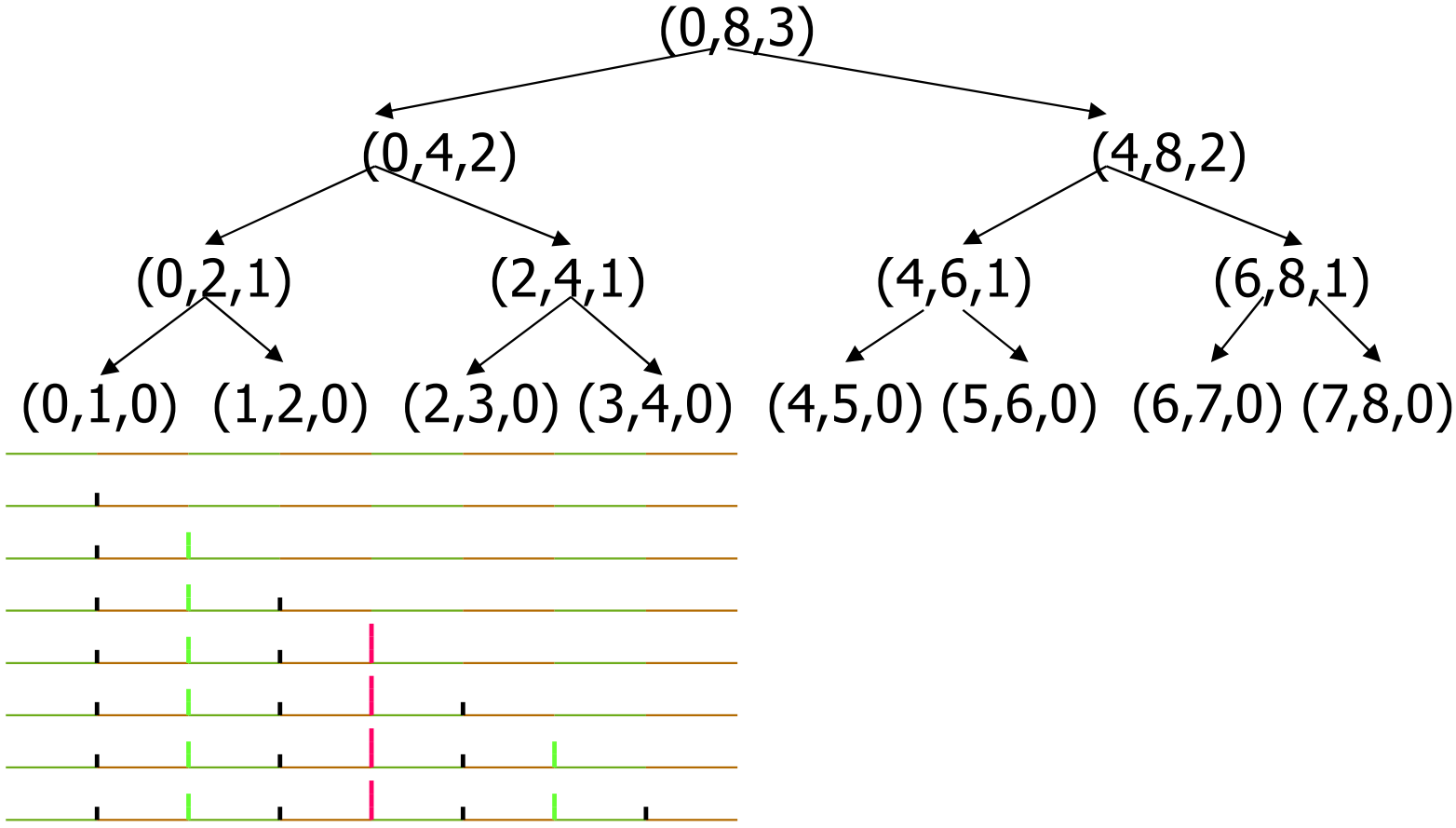
- la tacca centrale è alta n unità,
- le due tacche al centro delle due metà di destra e sinistra sono alte n-1
- etc.
- `mark(x, h)` traccia una tacca alta h unità in posizione x

```
void mark(int m, int h) {
  int i;
  printf("%d \t", m);
  for (i = 0; i < h; i++)
    printf("*");
  printf("\n");
}
```

Strategia divide et impera

- Dividiamo l'intervallo in due metà
- Disegniamo ricorsivamente le tacche (più corte) nella metà di SX
- Disegniamo la tacca (più lunga) al centro
- Disegniamo ricorsivamente le tacche (più corte) nella metà di DX
- Condizione di terminazione: tacche di altezza 0

Esempio



```
void ruler(int l, int r, int h) {
  int m;
  m = (l + r)/2;
  if (h > 0) {
    ruler(l, m, h-1);
    mark(m, h);
    ruler(m, r, h-1);
  }
}
```

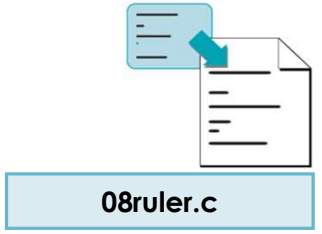
discesa ricorsiva/terminazione

chiamata ricorsiva

divisione

soluzione elementare

chiamata ricorsiva



08ruler.c

Analisi di complessità

- $D(n) = \Theta(1)$, $C(n) = \Theta(1)$
- $a = 2$, $b = 2$

Equazione alla ricorrenze:

$T(n) = 2T(n/2) + 1$ $\quad$ $n > 1$

$T(1) = 1$

$T(n) = O(n)$

Meccanismi computazionali per eseguire funzioni ricorsive

Chiamata a funzione: quando si chiama una funzione:

- si crea una nuova istanza della funzione chiamata
- si alloca memoria per i parametri e per le variabili locali
- si passano i parametri
- il controllo passa dal chiamante alla funzione chiamata
- si esegue la funzione chiamata
- al suo termine, il controllo ritorna al programma chiamante, che esegue l'istruzione immediatamente successiva alla chiamata a funzione.

È possibile che una funzione ne chiami un'altra. Serve un meccanismo per gestire le chiamate annidate e i relativi ritorni: lo **stack**.

Stack

- Definizione: tipo di dato astratto (ADT) che supporta operazioni di:
  - Push: inserimento dell'oggetto in cima allo stack
  - Pop: prelievo (e cancellazione) dalla cima dell'oggetto inserito più di recente
- Terminologia: la strategia di gestione dei dati è detta LIFO (Last In First Out)

Si chiama **stack frame** (o **record di attivazione**) la struttura dati che contiene almeno:

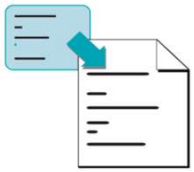
- i parametri formali
- le variabili locali
- l'indirizzo a cui si ritornerà una volta terminata l'esecuzione della funzione
- il puntatore al codice della funzione.

Lo stack frame viene creato alla chiamata della funzione e distrutto al suo termine.

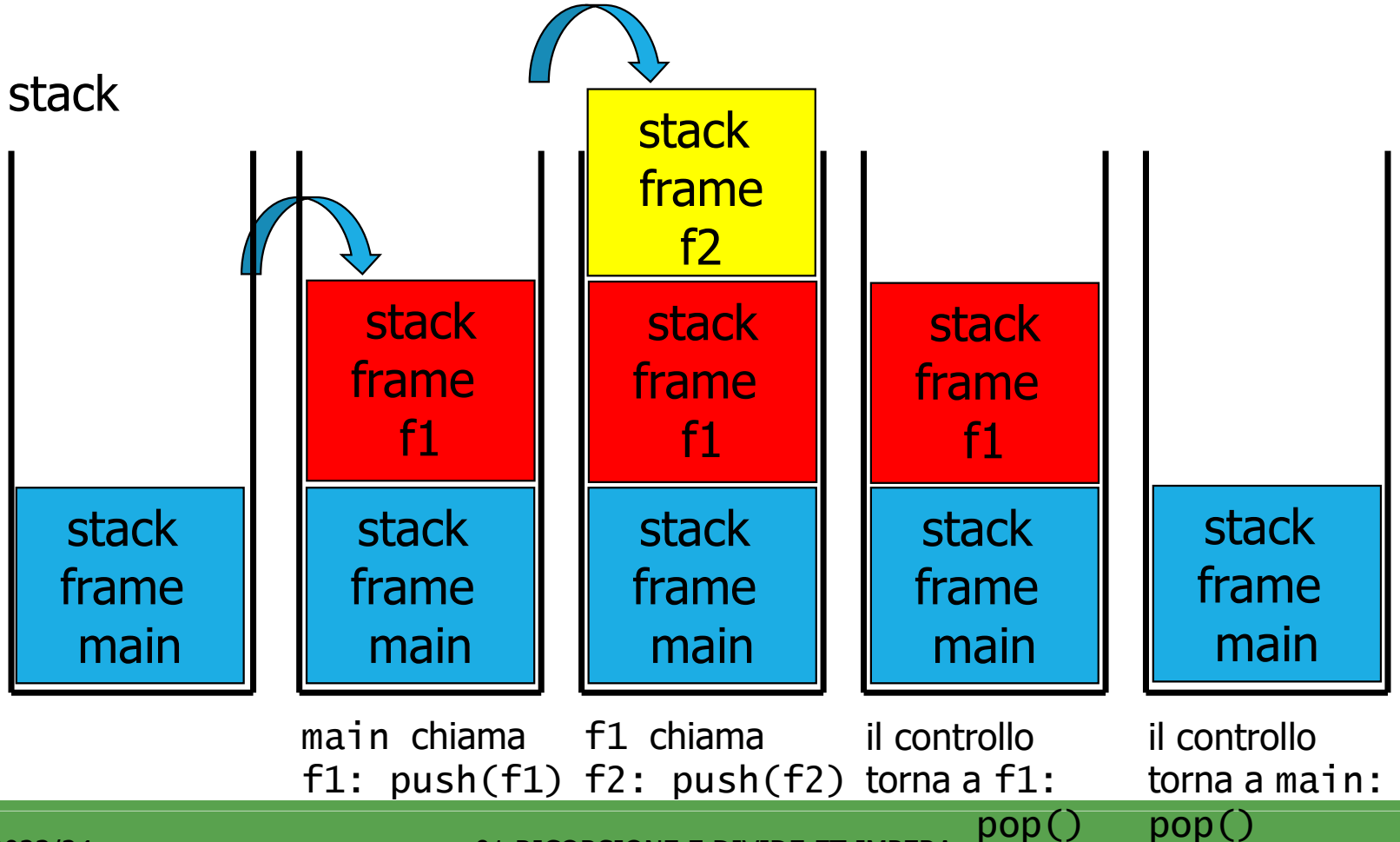
Gli stack frame sono memorizzati nello stack di sistema.
Lo stack di sistema ha a disposizione una quantità prefissata di memoria. Quando oltrepassa lo spazio allocatogli, c'è **stack overflow**.
Lo stack cresce da indirizzi maggiori a indirizzi minori (quindi verso l'alto). Lo **stack pointer SP** è un registro che contiene l'indirizzo del primo stack frame disponibile.

Esempio

```
int f1(int x);
int f2(int x);

main() {
  int x, a = 10;
  x = f1(a);
  printf("x is %d \n", x);
}

int f1(int x) { return f2(x); }

int f2(int x) { return x+1; }
```



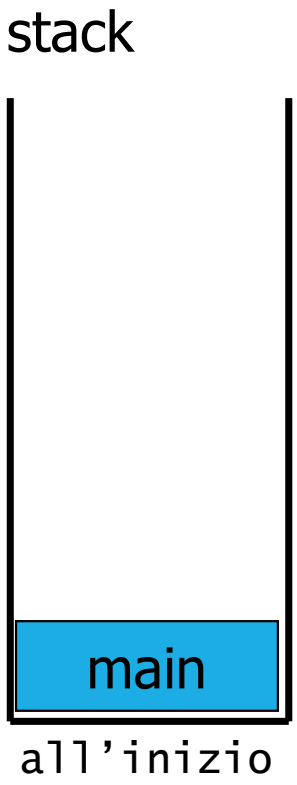
10function_call.c


stack
stack frame main
stack frame f1
stack frame main
stack frame f2
stack frame f1
stack frame main
stack frame f1
stack frame main
stack frame main
main chiama f1: push(f1)
f1 chiama f2: push(f2)
il controllo torna a f1: pop()
il controllo torna a main: pop()

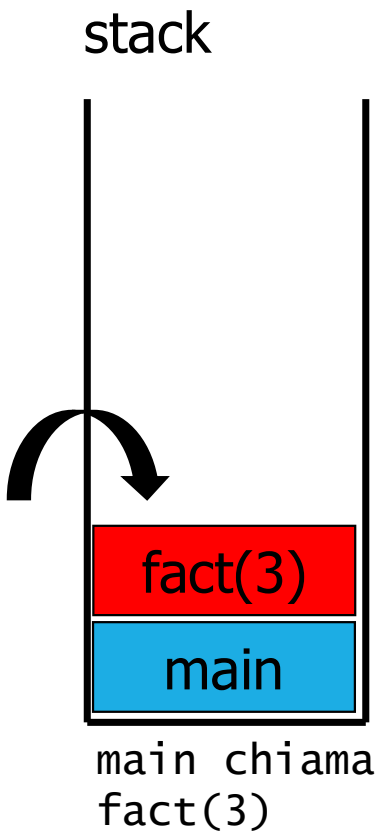
Funzioni ricorsive

- Funzione chiamante e chiamata coincidono, operano però su valori diversi
- Si usa lo stack di sistema come in una qualsiasi chiamata a funzione
- Un numero eccessivo di chiamate ricorsive può portare allo stack overflow.

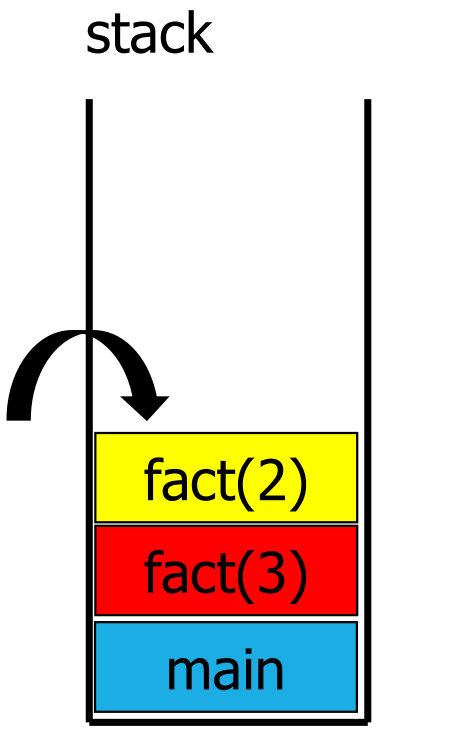
Esempio: calcolo di 3!




```
int main() {
  int n;
  printf("Input n (<=12):  ");
  scanf("%d", &n);
  printf("%d %lu \n",n, fact(n));
  return 0;
}
unsigned long fact(int n) {
  if(n == 0)
    return 1;
  return n*fact(n-1);
}
```



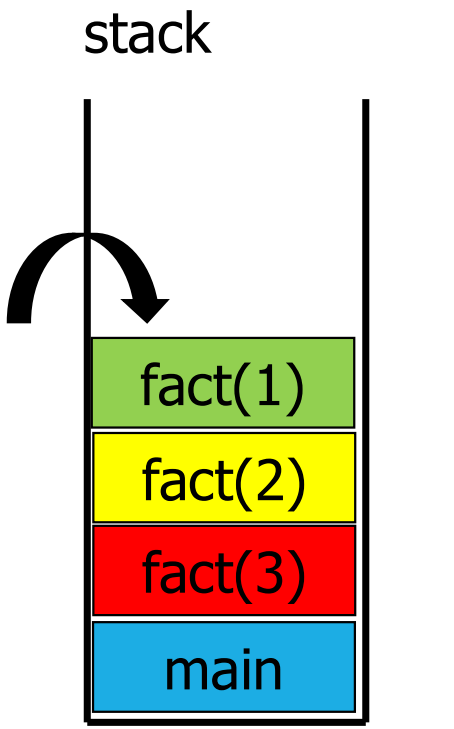
3! = 3*2!



fact(3) chiama
fact(2)

3! = 3*2!

2! = 2*1!

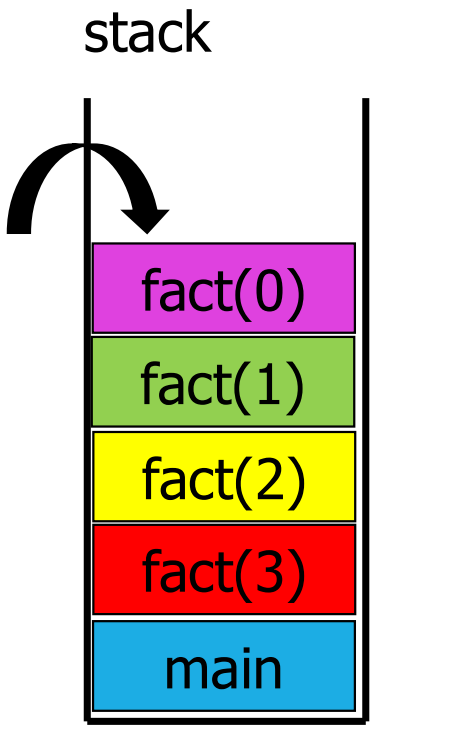


fact(2) chiama
fact(1)

3! = 3*2!

2! = 2*1!

1! = 1*0!



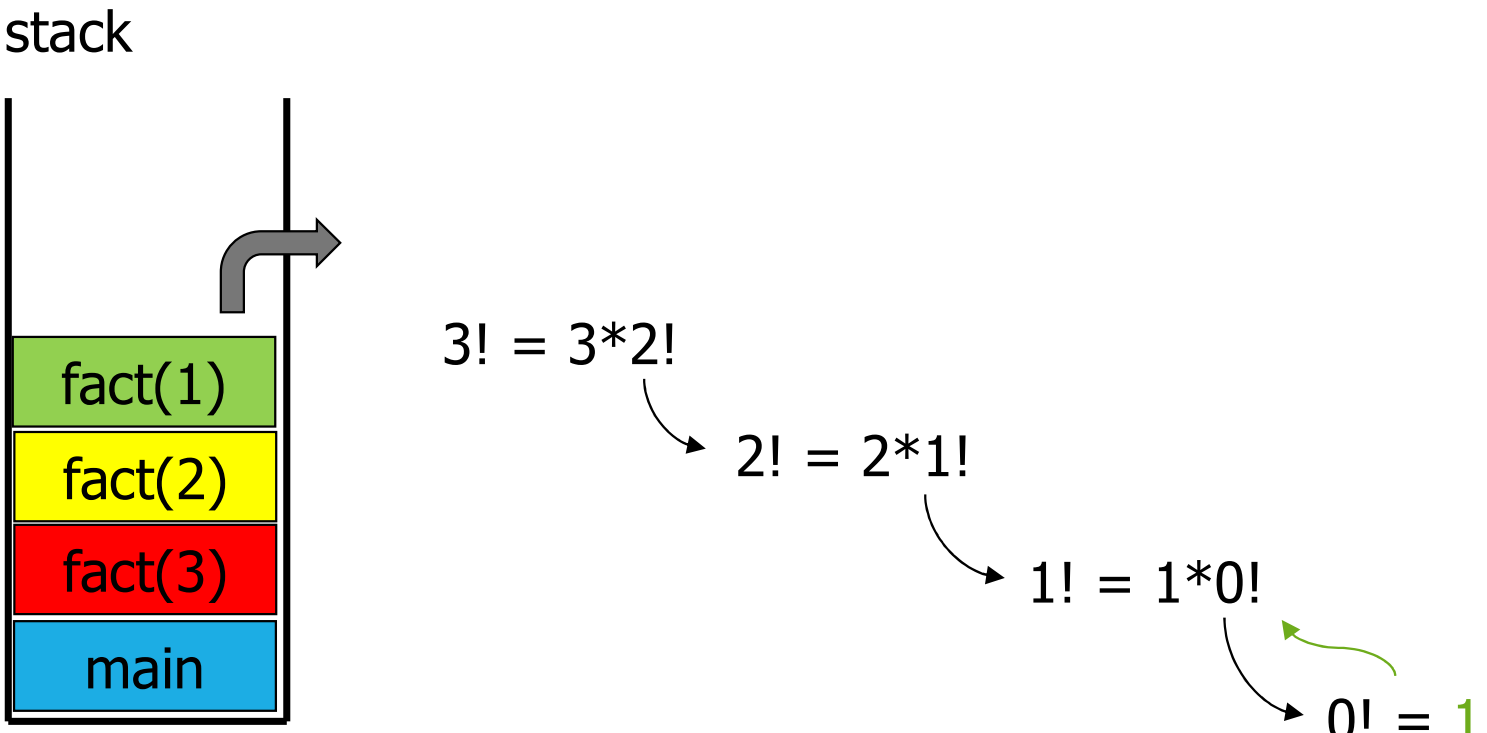

```
fact(1) chiama
fact(0)
```

3! = 3*2!

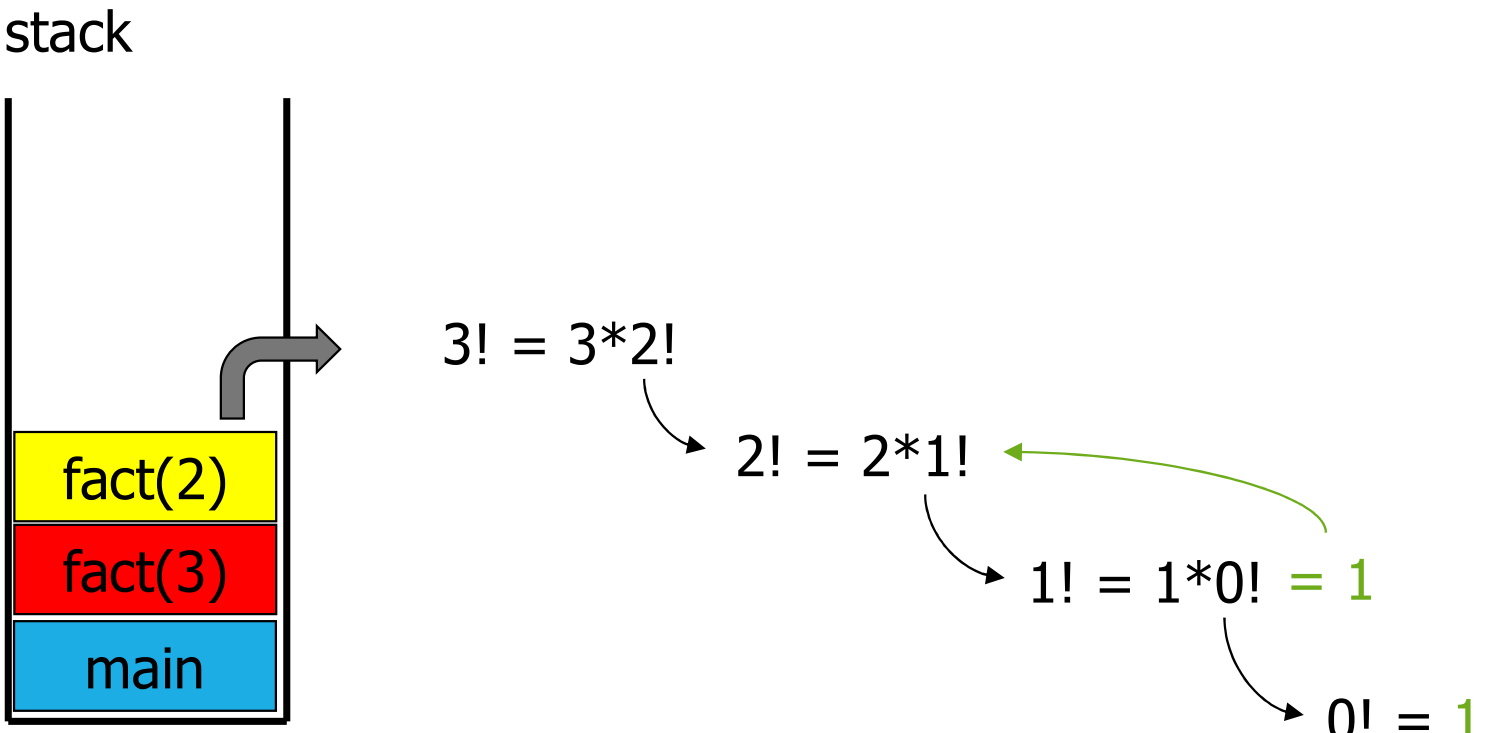
2! = 2*1!

1! = 1*0!

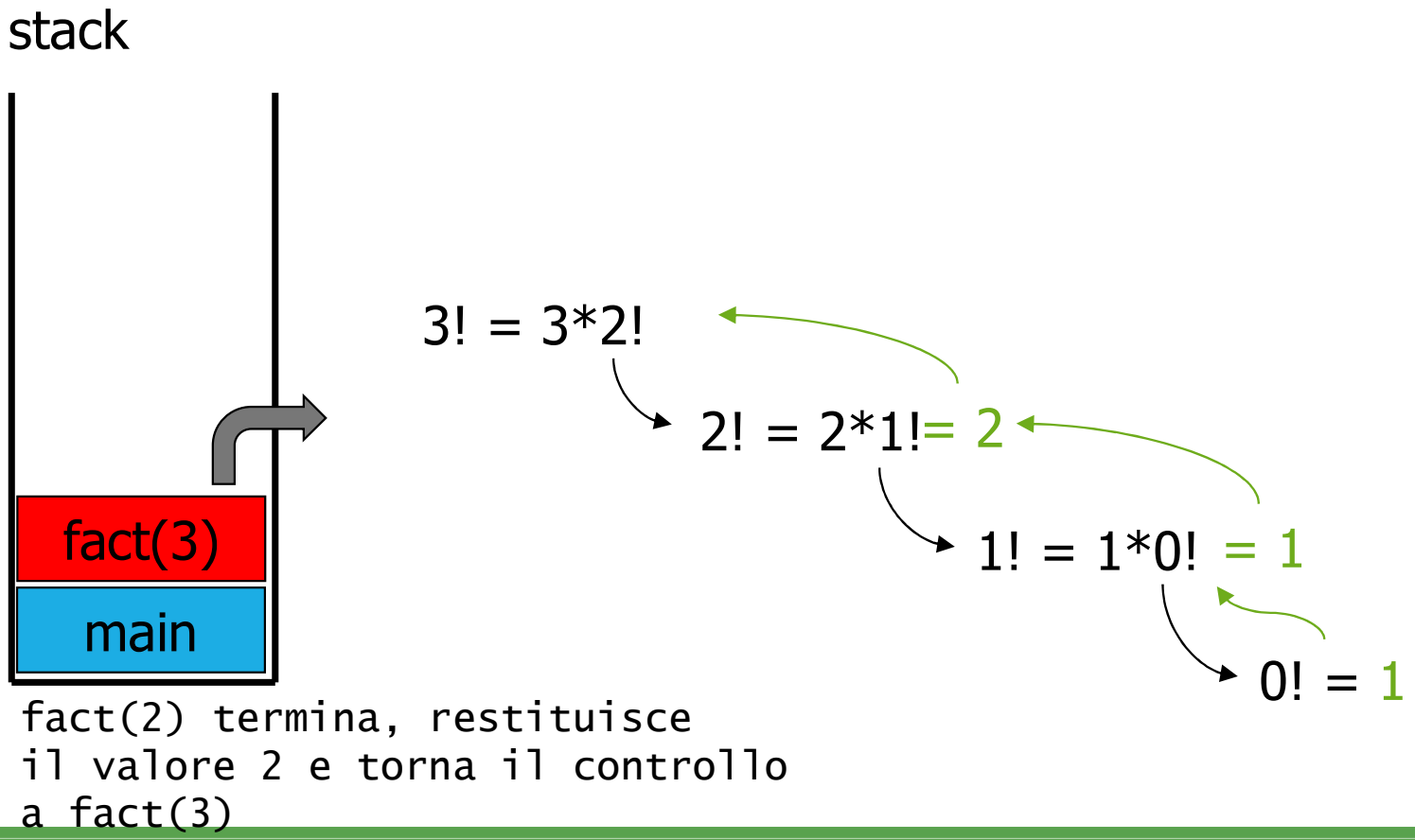
0! = 1



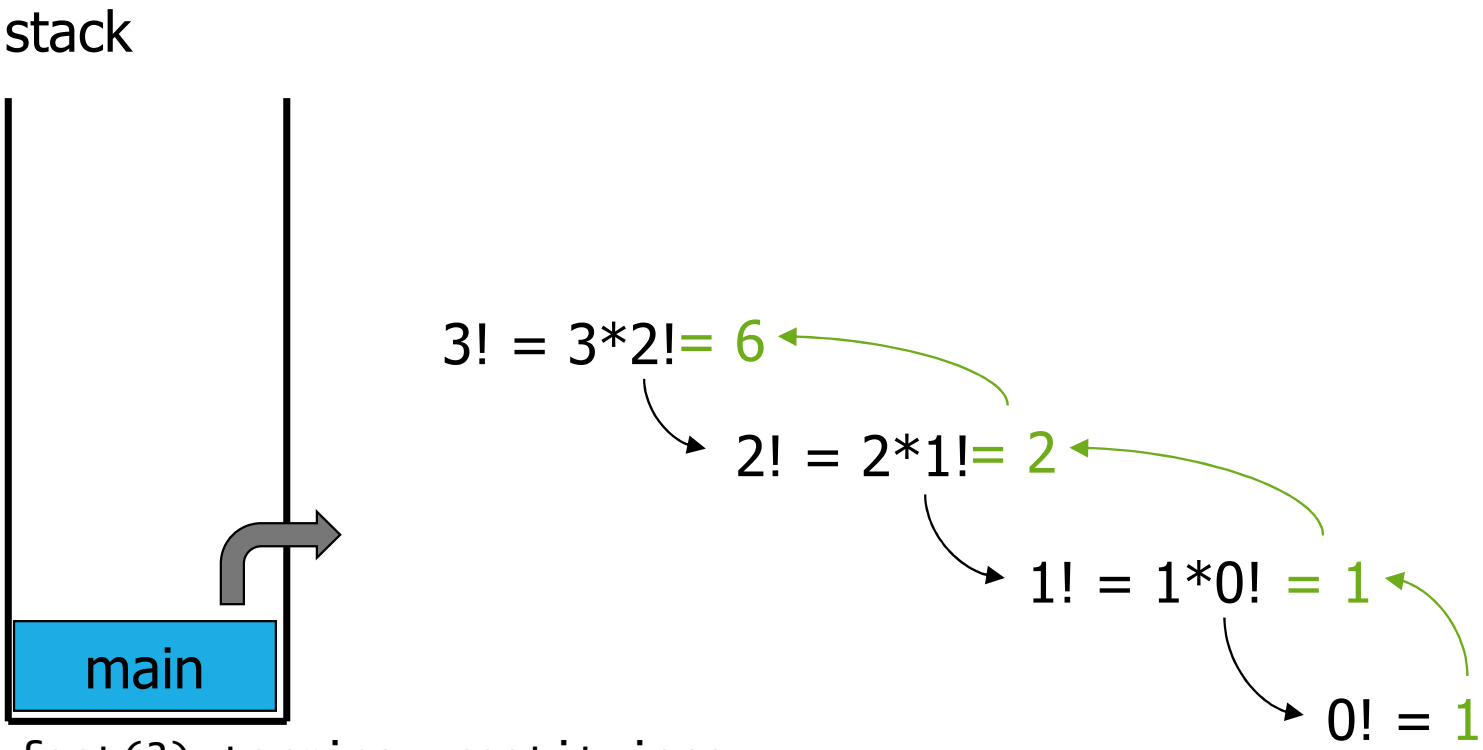
fact(0) termina, restituisce
il valore 1 e torna il controllo
a fact(1)



fact(1) termina, restituisce
il valore 1 e torna il controllo
a fact(2)

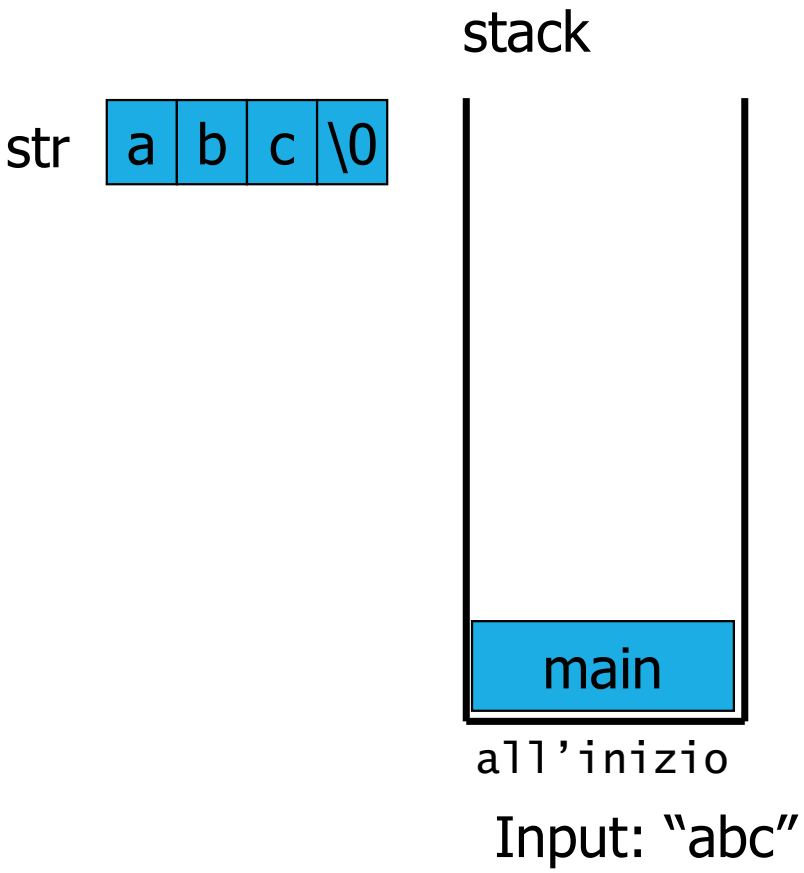


fact(2) termina, restituisce
il valore 2 e torna il controllo
a fact(3)

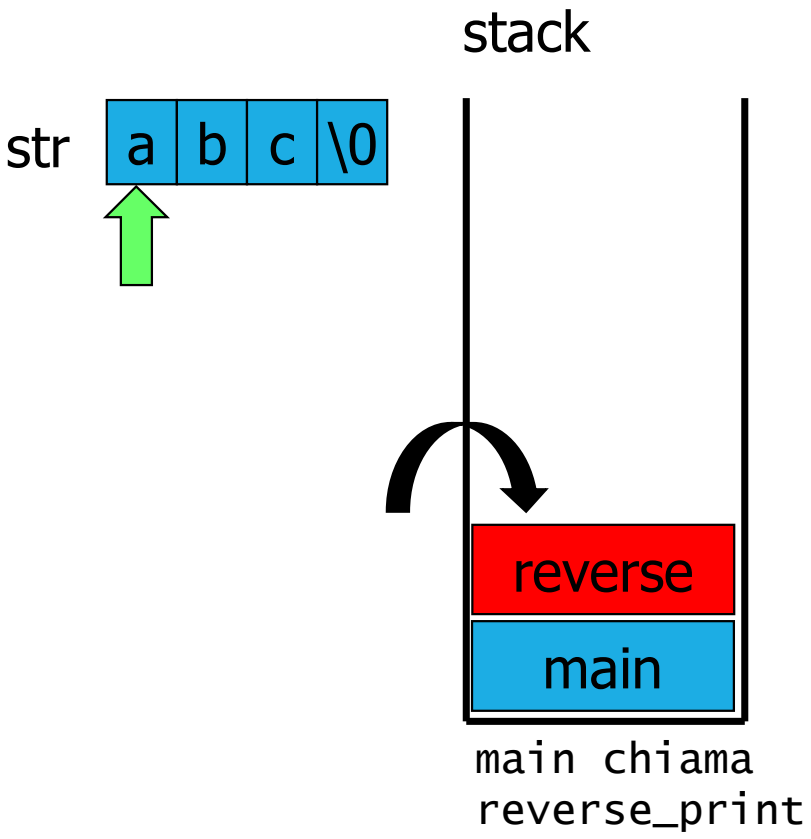


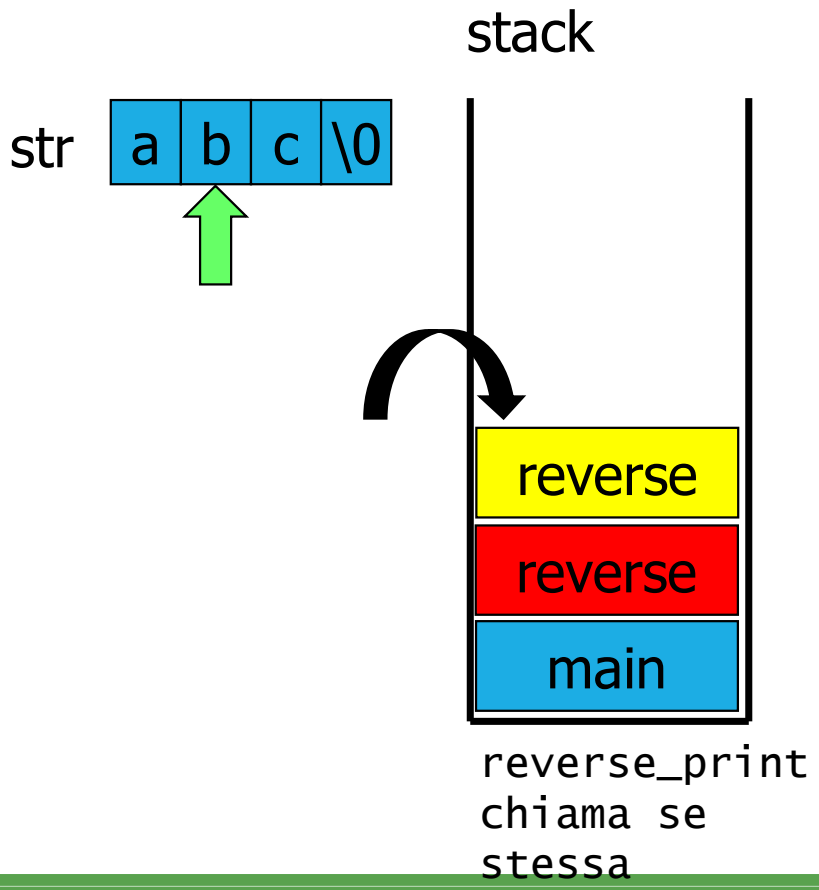
fact(3) termina, restituisce il valore 6 e torna il controllo al main

Esempio: reverse_print di “abc”

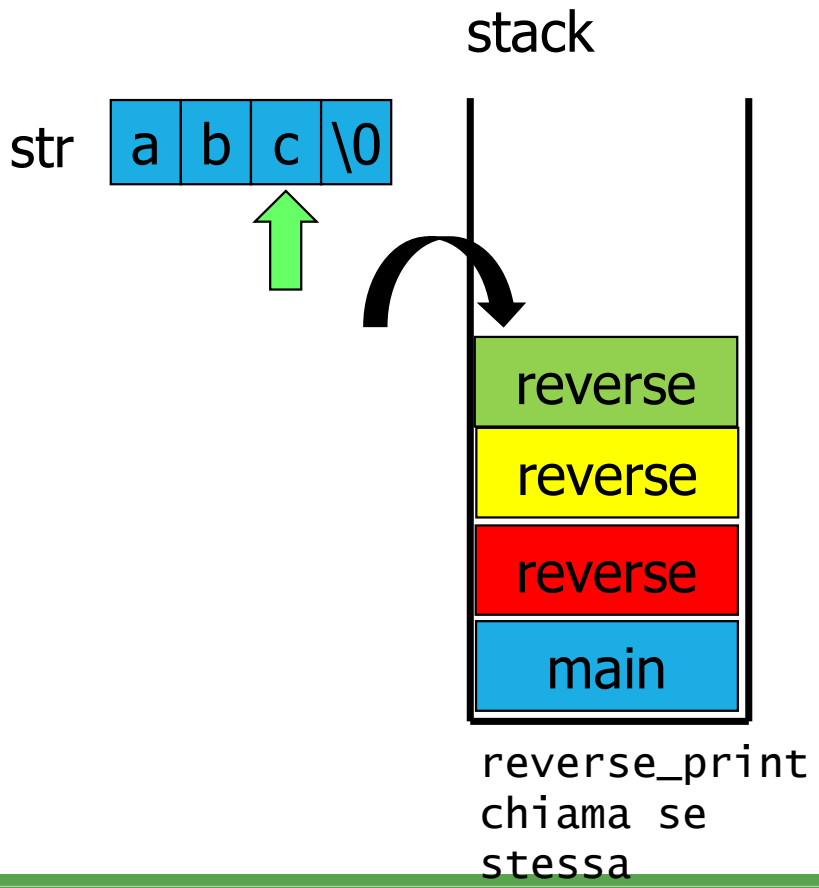



```
int main() {
  char str[max+1];
  printf("Input string: ");
  scanf("%s", str);
  printf("Reverse string is: ");
  reverse_print(str);
  return 0;
}

void reverse_print(char *s) {
  if(*s != '\0') {
    reverse_print(s+1);
    putchar(*s);
  }
  return;
}
```

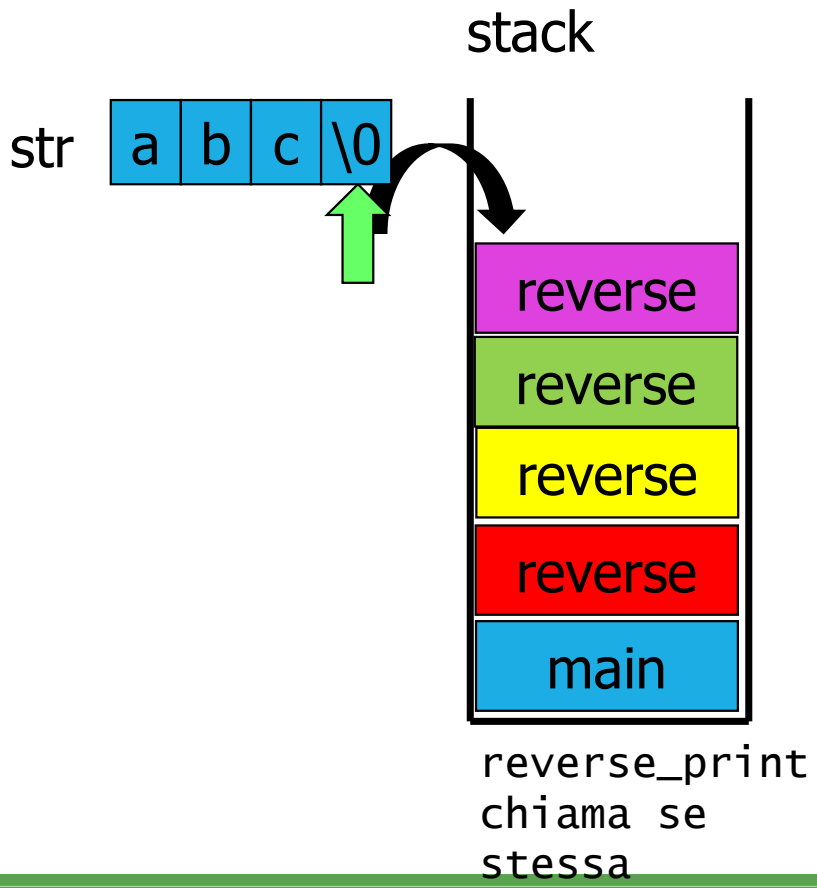

stack
str
a
b
c
\0
reverse
main
main chiama
reverse_print



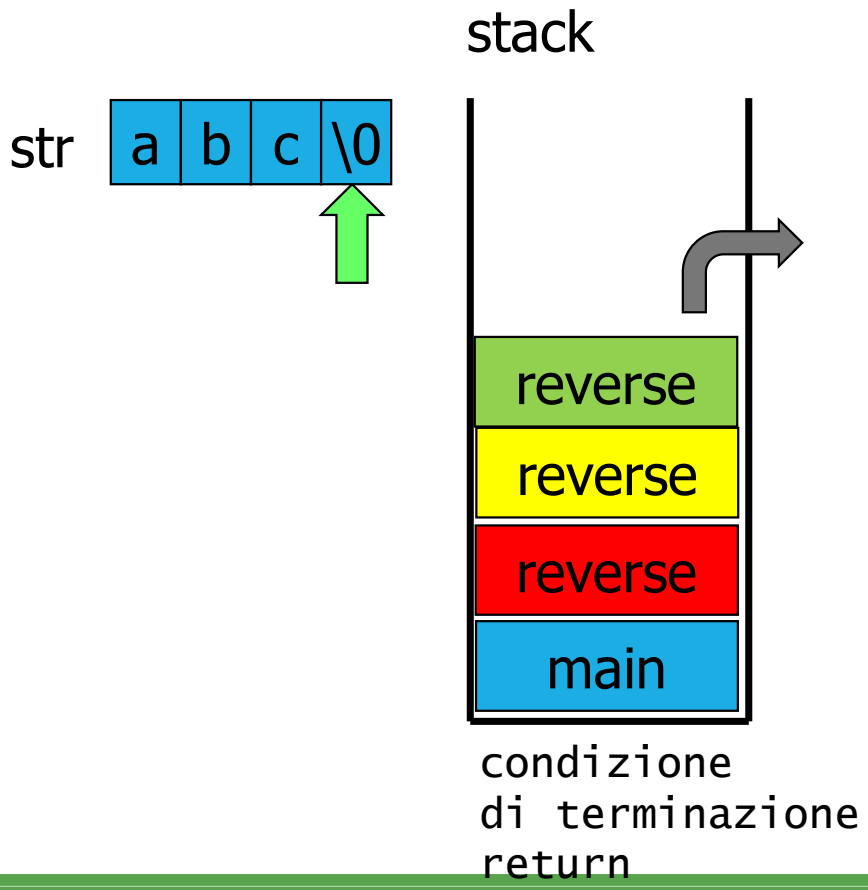

```
reverse_print
chiama se
stessa
```



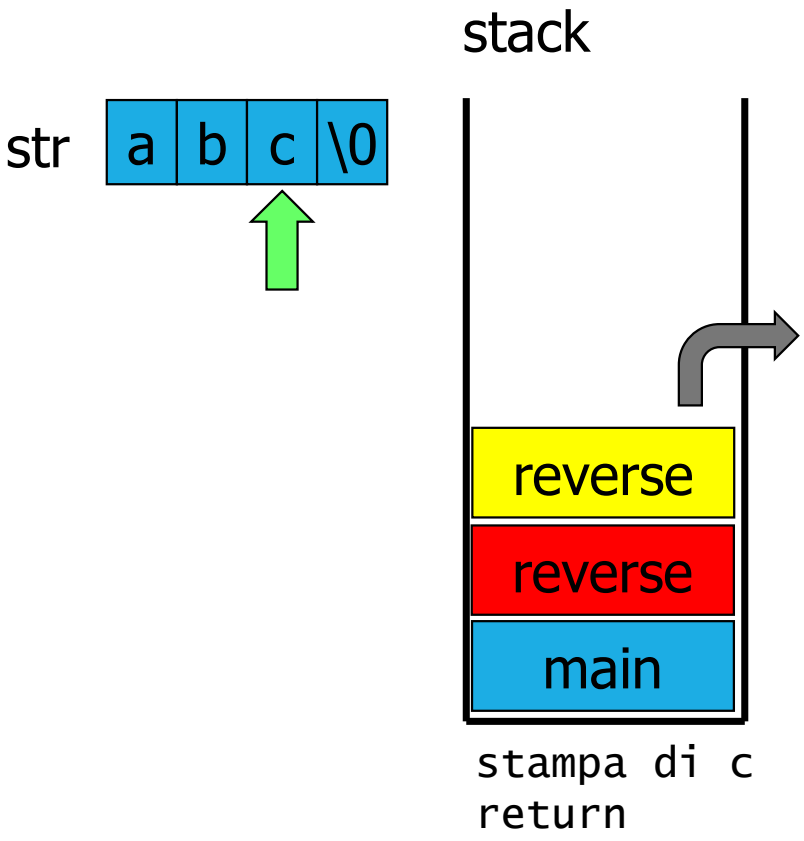
reverse_print
chiama se
stessa



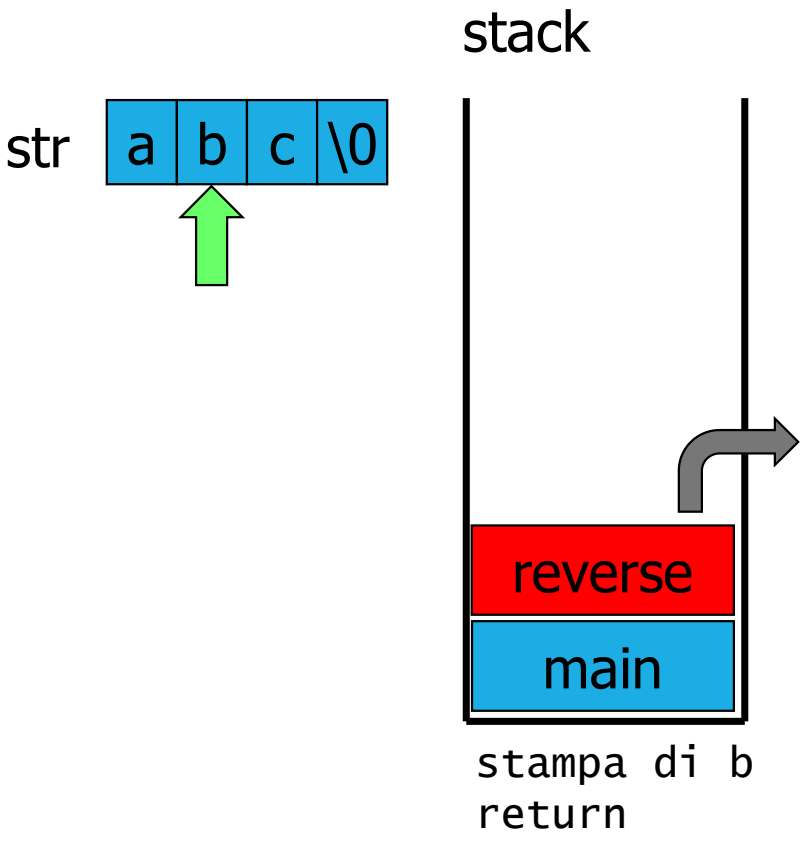
reverse_print
chiama se
stessa



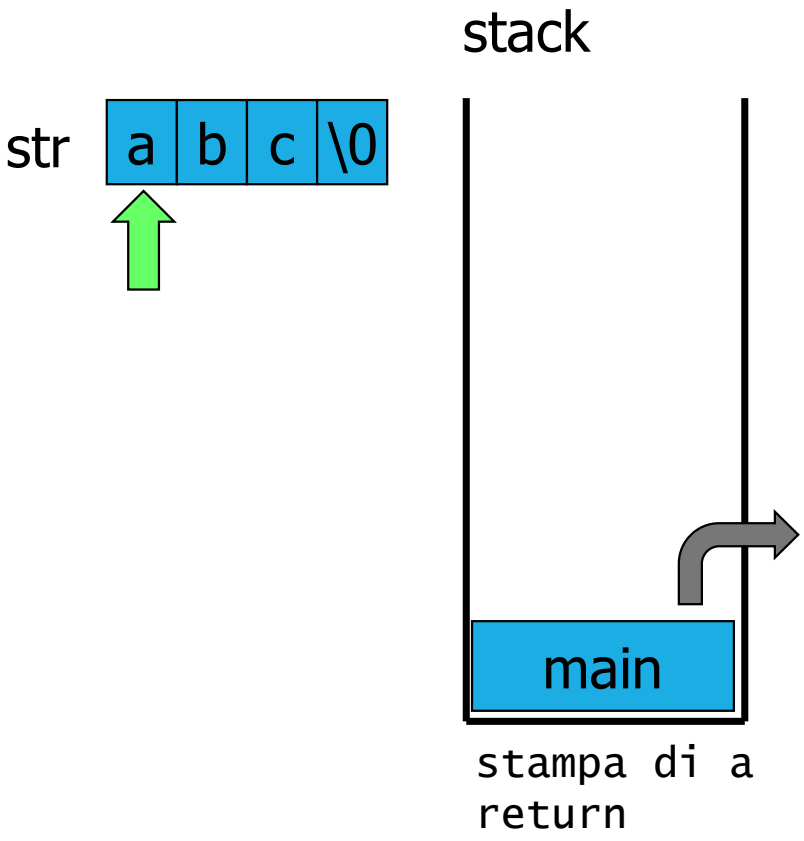

```
condizione
di terminazione
return
```



Output: “c”



Output: “cb”



Output: “cba”

Funzioni non tail-recursive

Una funzione ricorsiva non è tail-recursive se la chiamata ricorsiva non è l'ultima operazione da eseguire

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

la moltiplicazione può essere eseguita solo dopo il ritorno della chiamata ricorsiva

i calcoli si fanno in fase di risalita

discesa

risalita

```
fact(3)
3 * fact(2)
3 * (2 * fact(1))
3 * (2 * (1 * fact(0)))
3 * (2 * (1 * 1))
```

```
3 * (2 * 1)
3 * 2
6
```

È necessario mantenere uno stack!

Funzioni tail-recursive

Nel main:

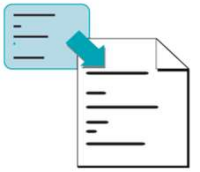
```
result = tr_fact(n, 1);
```

Una funzione ricorsiva è tail-recursive se la chiamata ricorsiva è l'ultima operazione da eseguire, eccezion fatta per `return`

```
unsigned long tr_fact(int n, unsigned long f)
{
  if(n == 0)
    return f;
  return tr_fact(n-1, n*f);
}
```

è tail-recursive perché la moltiplicazione viene eseguita prima della chiamata ricorsiva

i calcoli si fanno in fase di discesa



11tail_recursive_factorial.c

discesa

```
tr_fact(3,1)
tr_fact(2,3)
tr_fact(1,6)
tr_fact(0,6)
```

Se una funzione è tail-recursive la funzione chiamante (caller) deve solo ritornare il valore calcolato dalla funzione chiamata (callee)

⇓

l'operazione di pop dello stack frame della funzione chiamante avviene prima dell'operazione di push nello stack frame della funzione chiamata

⇓

lo stack frame della funzione chiamata rimpiazza semplicemente quello della funzione chiamante

⇓

l'occupazione di memoria non è più O(n) dove n sono i livelli di ricorsione, bensì O(1)

⇓

non ci può più essere stack overflow

Svantaggi:

- più difficili da scrivere
- non tutti i compilatori (tra cui quello del C) riescono a sfruttare le funzioni tail-recursive per ottimizzare il codice.

Dualità ricorsione - iterazione

Possibili soluzioni:

- iterativa “nativa”
- ricorsiva:
  - se tail-recursive trasformabile direttamente in iterativa senza uso di stack
  - se non tail-recursive trasformabile in iterativa con uso di stack. In generale soluzione meno efficienti di quelle iterative “native”.

direttamente in forma iterativa

Fibonacci:

$FIB_0 = 0$

$FIB_1 = 1$

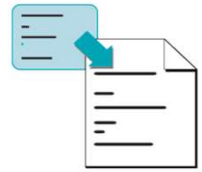
$FIB_2 = FIB_0 + FIB_1 = 1$

$FIB_3 = FIB_1 + FIB_2 = 2$

$FIB_4 = FIB_2 + FIB_3 = 3$

$FIB_5 = FIB_3 + FIB_4 = 5$

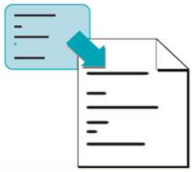
```
unsigned long fib(int n) {
  unsigned long f1p=1, f2p=0, f;
  int i;
  if(n == 0 || n == 1)
    return n;
  f = f1p + f2p; /* n==2 */
  for(i=3; i<= n; i++) {
     f2p = f1p;
     f1p = f;
     f = f1p+f2p;
   }
 return f;
}
```



12iterative_fibonacci.c

Ricerca binaria

```
int BinSearch(int v[], int N, int k) {
  int m, found= 0, l=0, r=N-1;
while(l <= r && found == 0){
    m = (l+r)/2;
    if(v[m] == k)
      found = 1;
    if(v[m] < k)
      l = m+1;
    else
      r = m-1;
  }
  if (found == 0)
    return -1;
  return m;
}
```

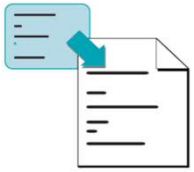


13iterative_binsearch.c

direttamente in forma iterativa

Fattoriale

```
unsigned long fact(int  n) {
  unsigned long tot = 1;
  int i;

  for (i=2; i<=n; i++)
    tot = tot * i;

  return tot;
}
```

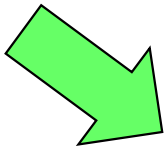


14iterative_factorial.c

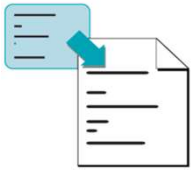
da tail-recursive a forma iterativa

Fattoriale

```
unsigned long tr_fact(int n, unsigned long f) {
  if(n == 0)
    return f;
  return tr_fact(n-1, n*f);
}
```



```
unsigned long tr2iterfact(int  n, unsigned long f){
  while (n > 0) {
    f = n * f;
    n--;
  }
  return f;
}
```



15tr2iterfact.c

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

da non tail-recursive
a iterativo con stack
gestito dall'utente

ADT S (stack)

```
unsigned long fact (int n){
  unsigned long f = 1;
  S stack;
  stack = STACKinit(N);

  while (n>0) {
    STACKpush(stack, n);
    n--;
  }
  while (STACKsize(stack) > 0) {
    n = STACKpop(stack);
    f = n * f;
  }
  return f;
}
```

Riferimenti

- Ricorsione
  - Sedgewick 5.1
  - Deitel 5.14, 5.15
- Divide et Impera
  - Sedgewick 5.2
  - Cormen 1.3.1
- Risoluzione di Equazioni alle Ricorrenze:
  - Cormen 4.2
- Algoritmo di Karatsuba:
  - Crescenzi 2.5.2
- Chiamata di funzioni
  - Deitel 5.7