Titolo unità didattica: Approccio divide et impera [13]

Titolo modulo : Algoritmi di raddoppiamento [03-T]

Algoritmi divide et impera per somma e massimo di un array

Argomenti trattati:

- ✓ algoritmo divide et impera per la somma di elementi di array
- ✓ algoritmo divide et impera di determinazione del massimo
- ✓ complessità di tempo di algoritmi divide et impera

Prerequisiti richiesti: P1-13-01-T

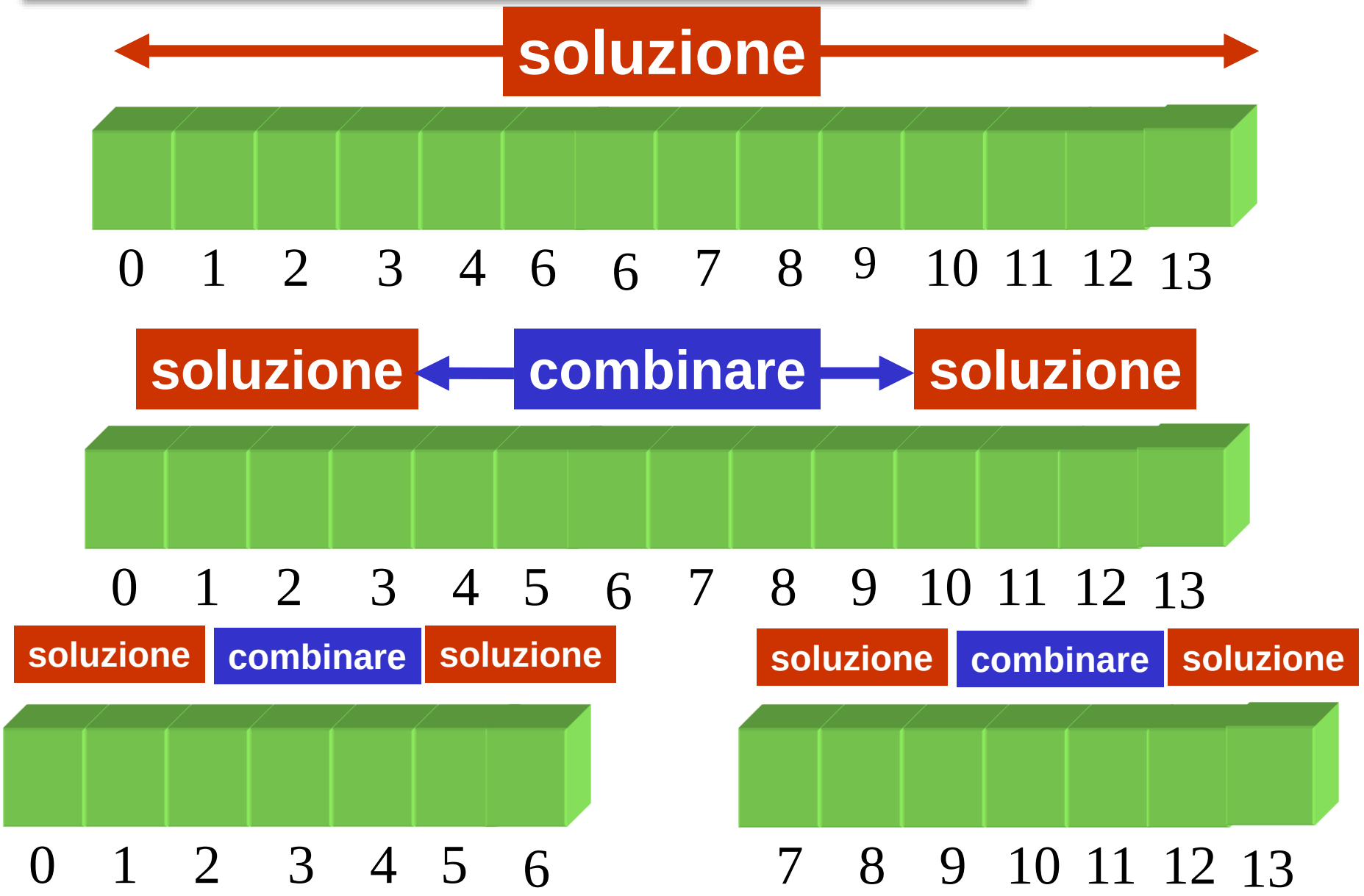
idea!

partendo dell'istanza da risolvere, si genera una **sequenza di istanze** via via più **semplici** del problema, fino all'istanza che non è più suddivisibile e che ha **soluzione banale**

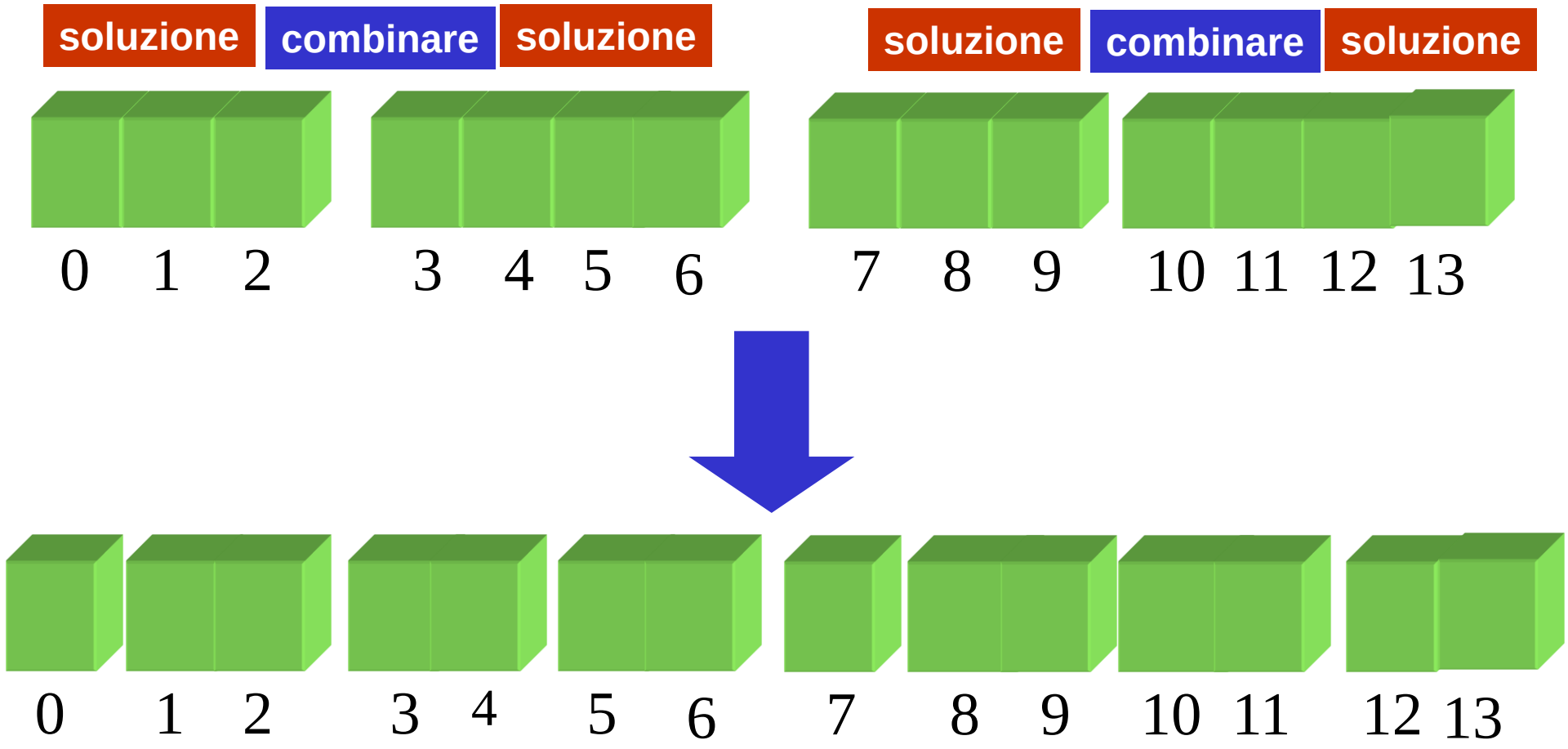
approccio divide et impera
(divide and conquer)

a ogni passo, la **soluzione dell'istanza** da risolvere viene espressa in termini delle **soluzioni delle istanze più semplici** in cui essa è **decomposta**

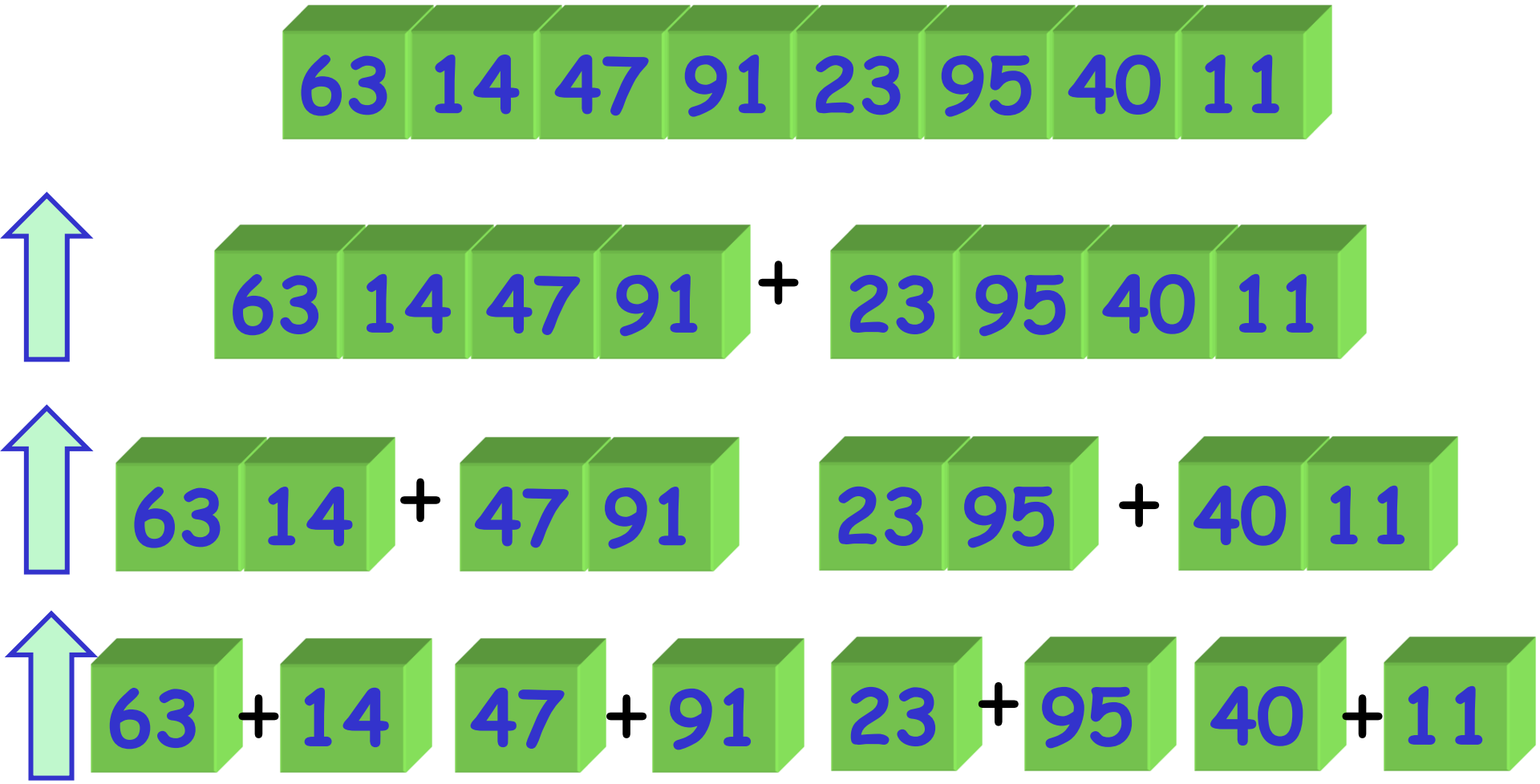
idea approccio divide et impera



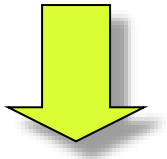
idea approccio divide et impera



approccio **divide et impera** al problema della **somma degli elementi** di un array di size n



calcolo della somma di n numeri



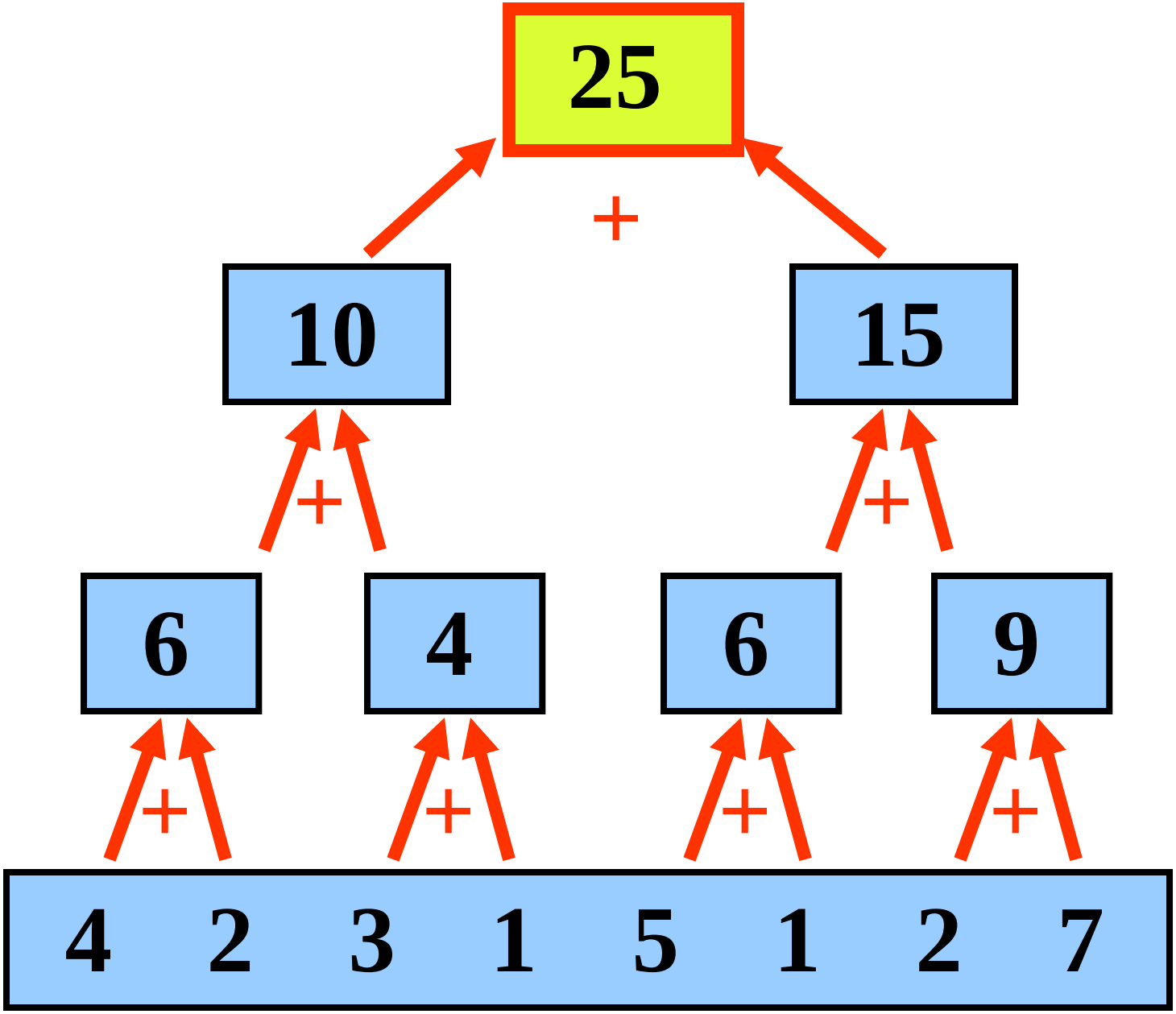
risoluzione di $n/2$ problemi di somma di coppie di numeri

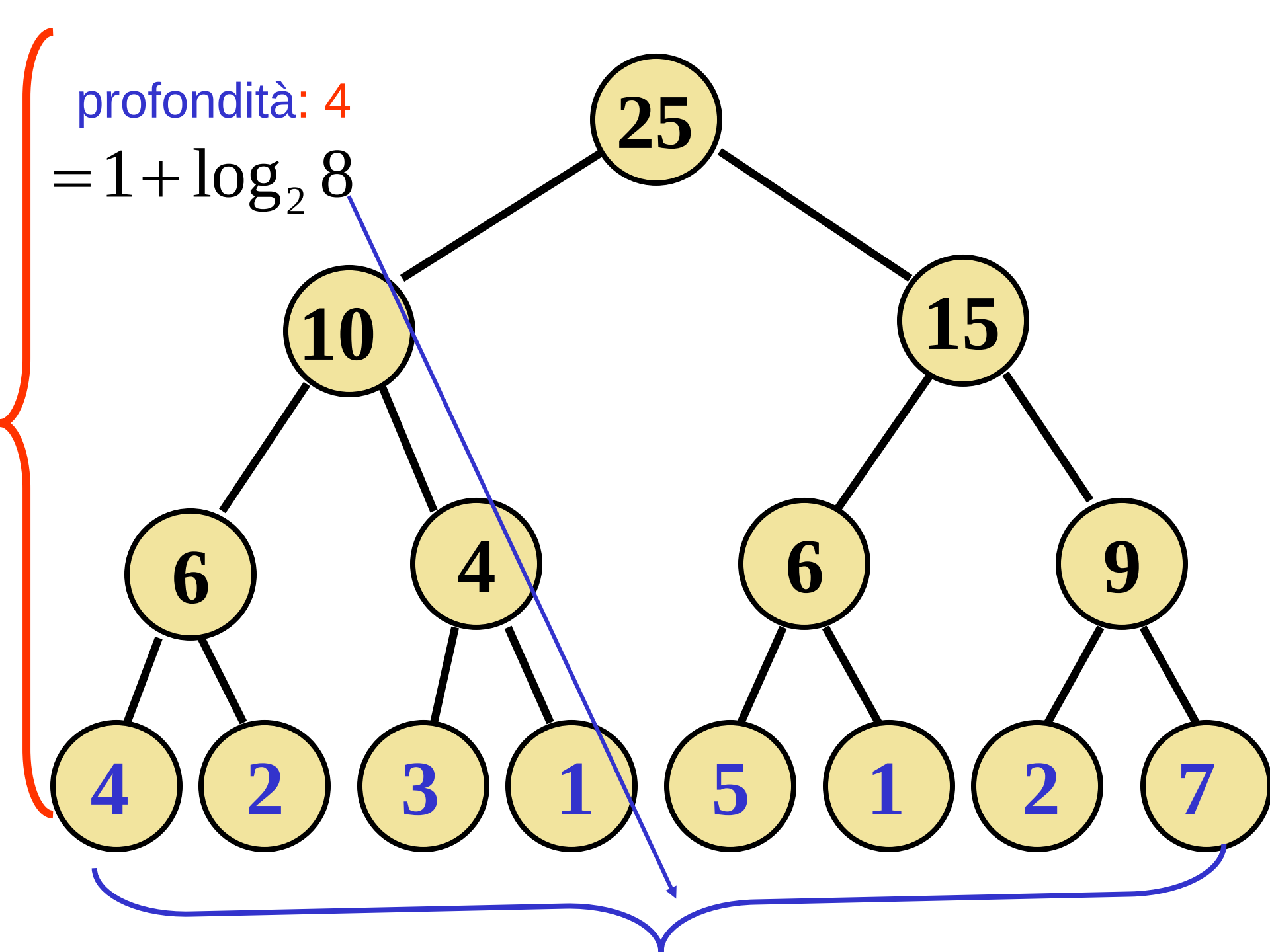
risoluzione di $n/4$ problemi di somma di coppie di numeri

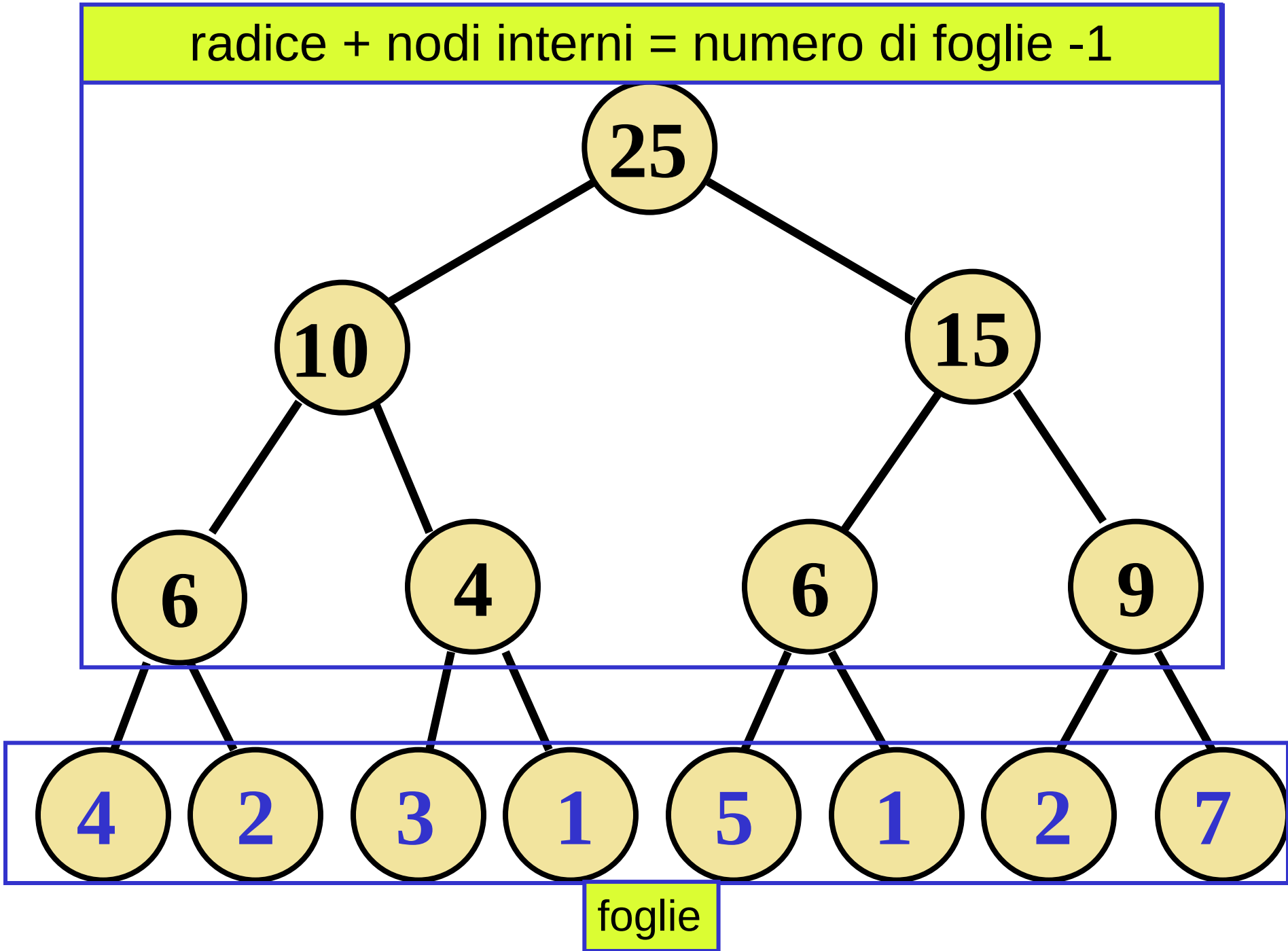
....

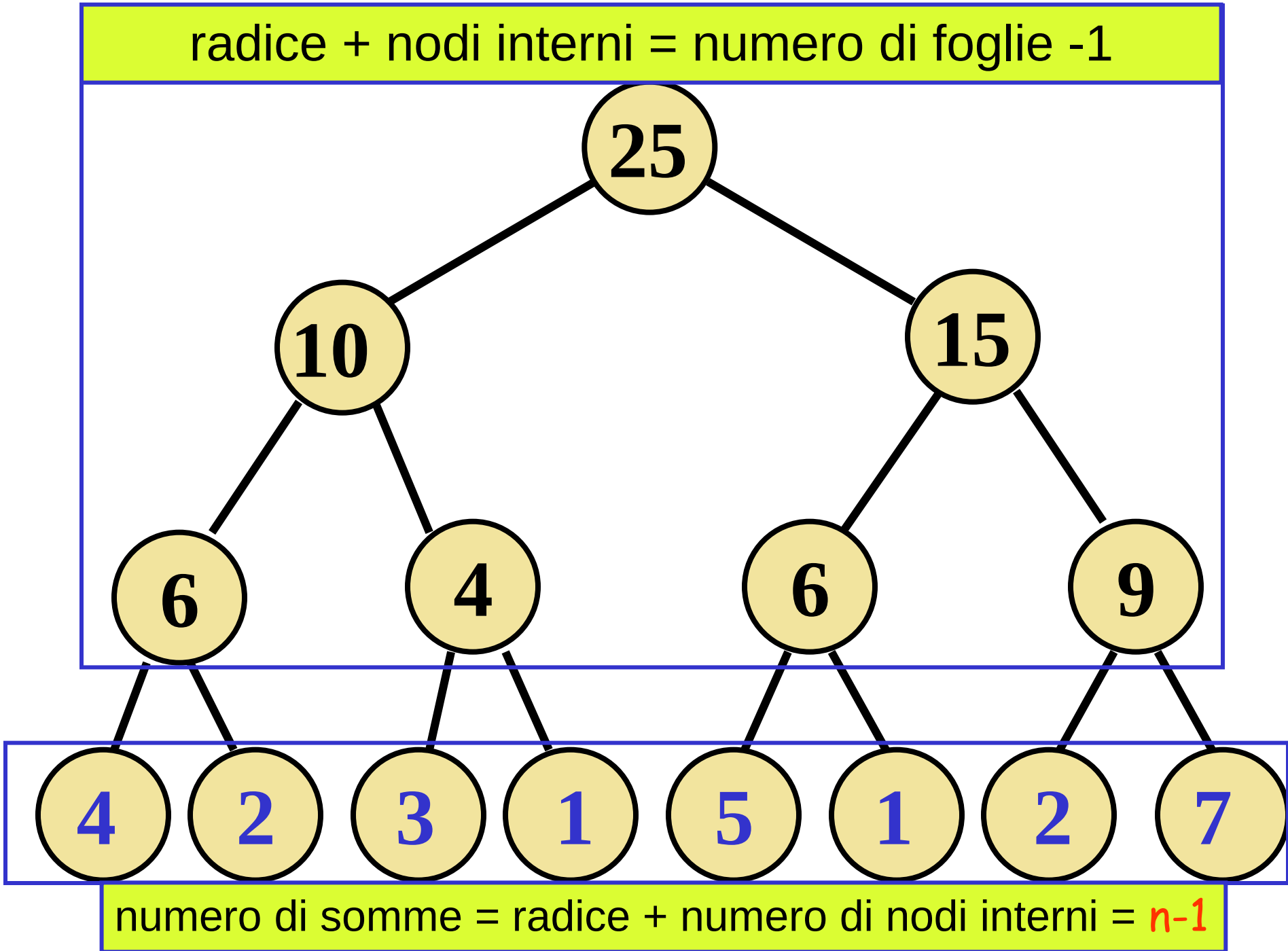
risoluzione di 1 problema di somma di coppie di numeri

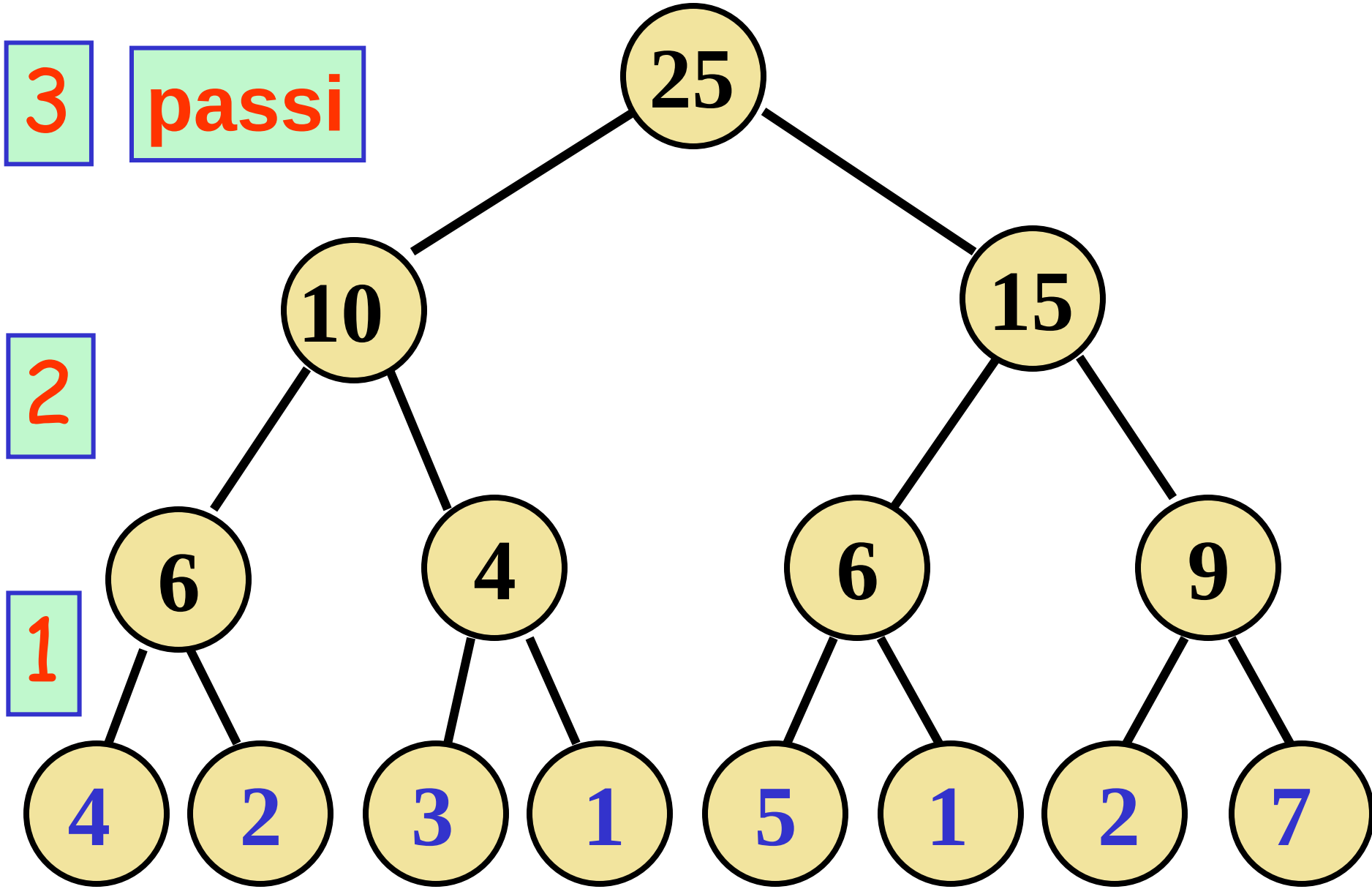
algoritmo di raddoppiamento

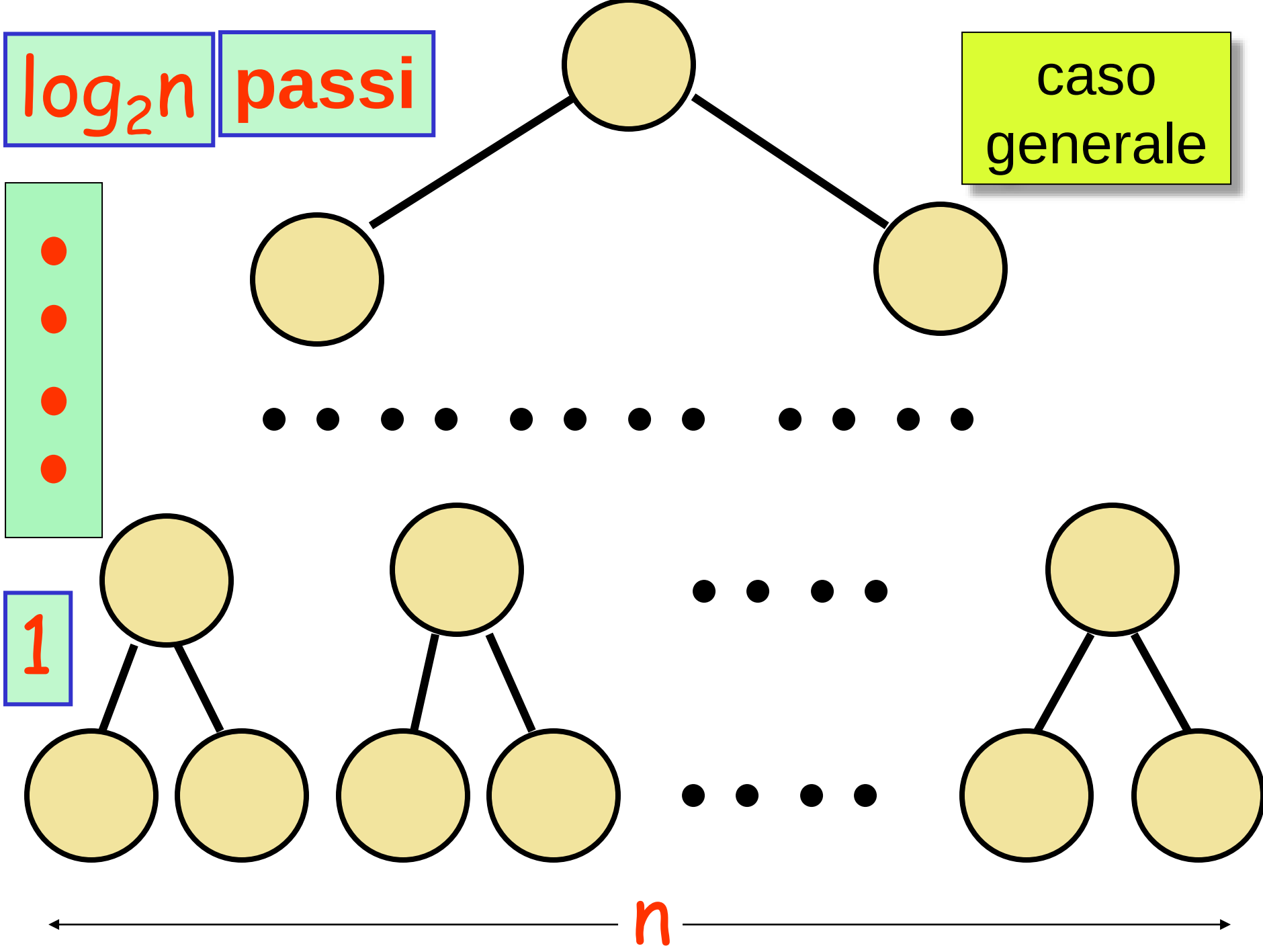

25
+
10
15
+
+
6
4
6
9
+
+
+
+
4 2 3 1 5 1 2 7

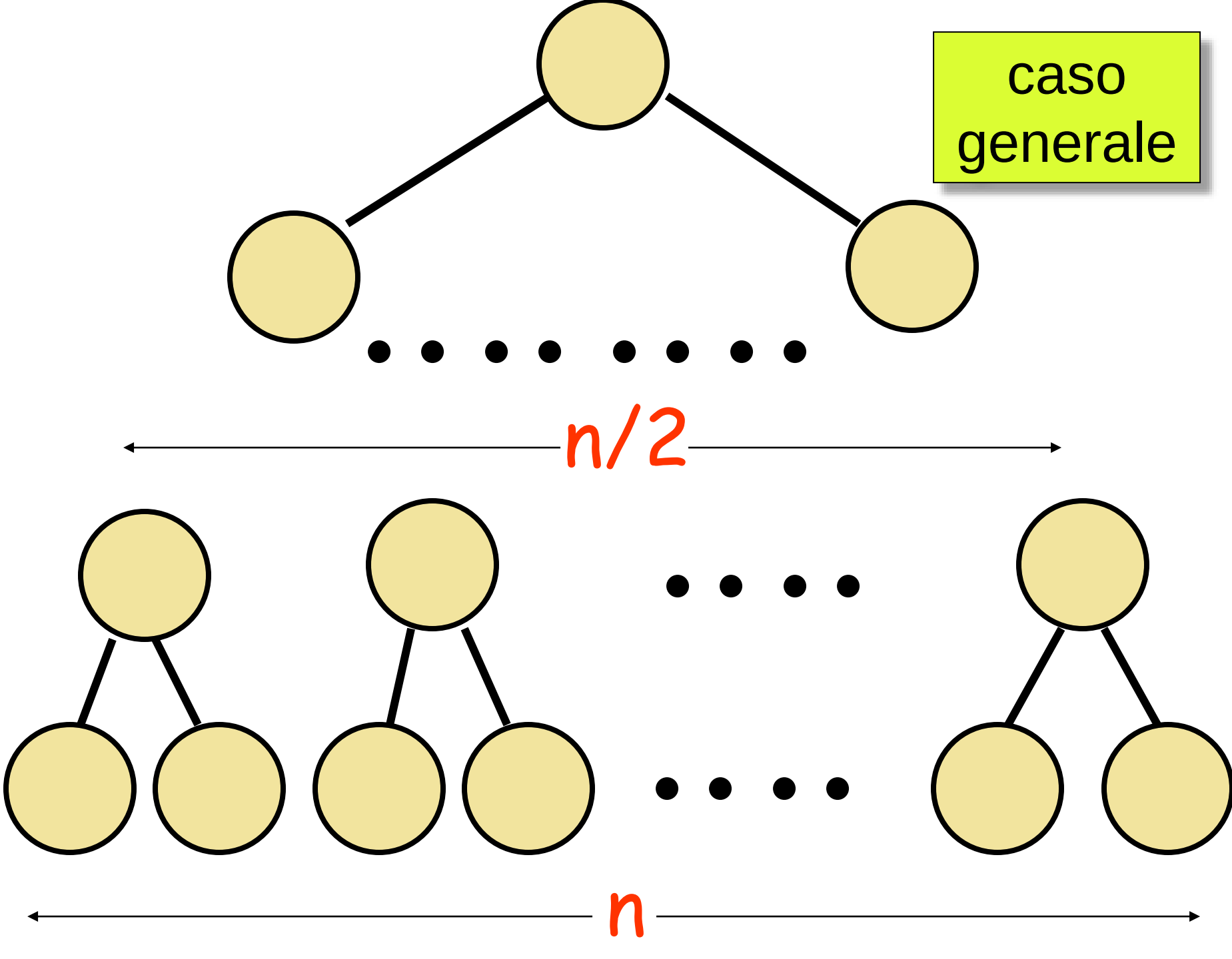

profondità: 4
$= 1 + \log_2 8$
25
10
15
6
4
6
9
4
2
3
1
5
1
2
7

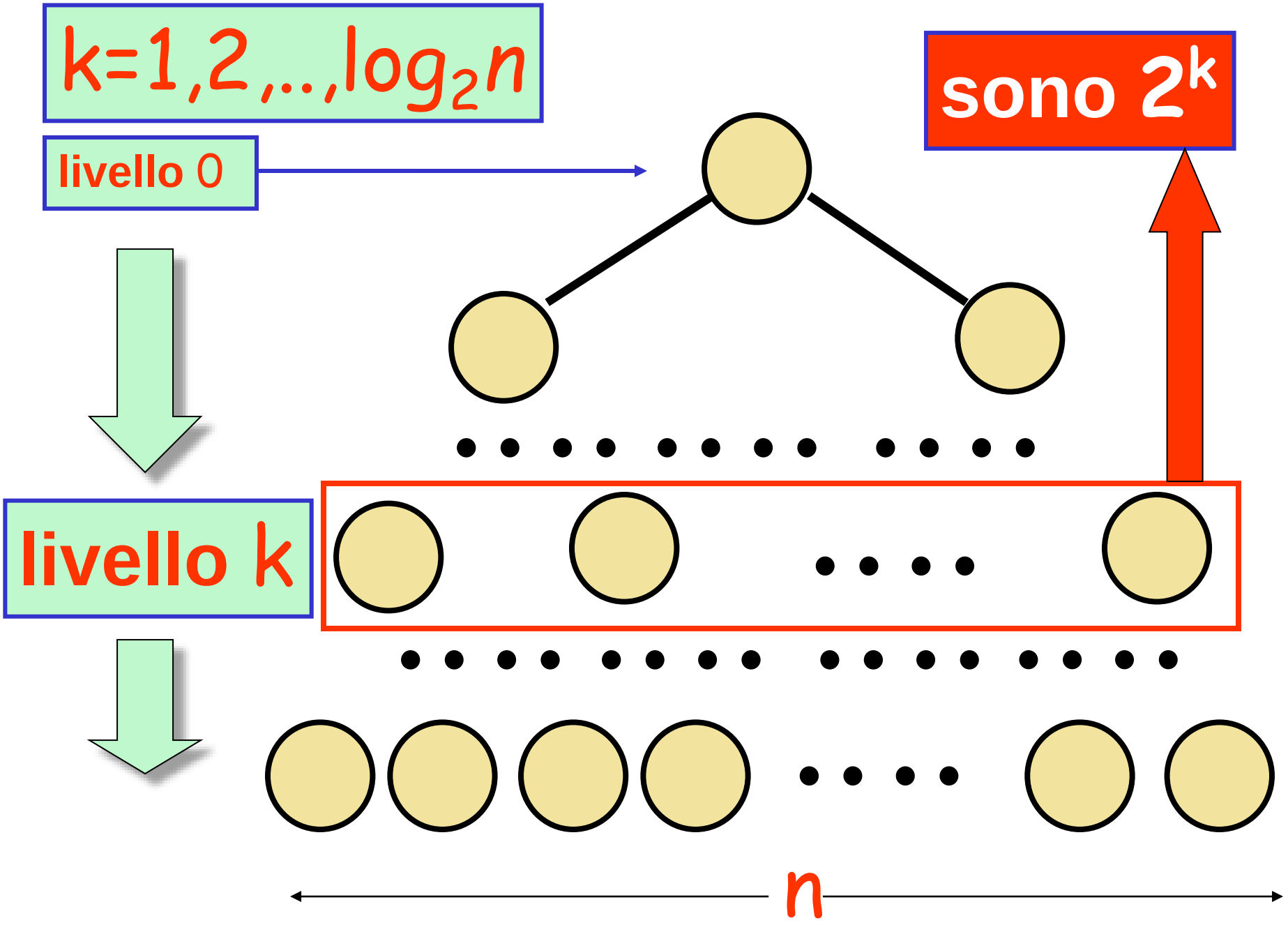

radice + nodi interni = numero di foglie -1
25
10
15
6
4
6
9
4
2
3
1
5
1
2
7
foglie

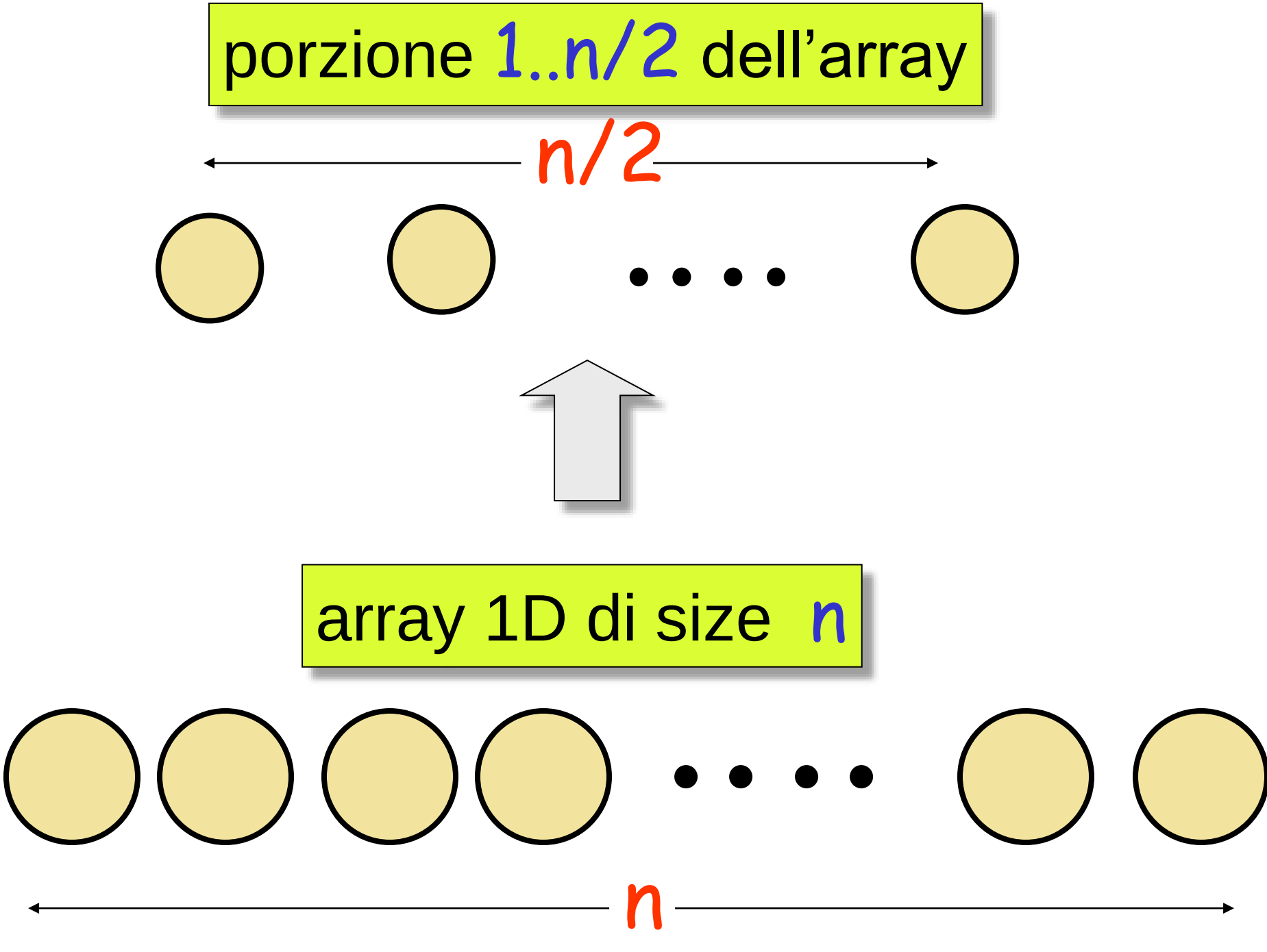

radice + nodi interni = numero di foglie -1
25
10
15
6
4
6
9
4
2
3
1
5
1
2
7
numero di somme = radice + numero di nodi interni = n-1


3
passi
2
1
25
10
15
6
4
6
9
4
2
3
1
5
1
2
7


$\log_2 n$ passi
caso generale
1
n

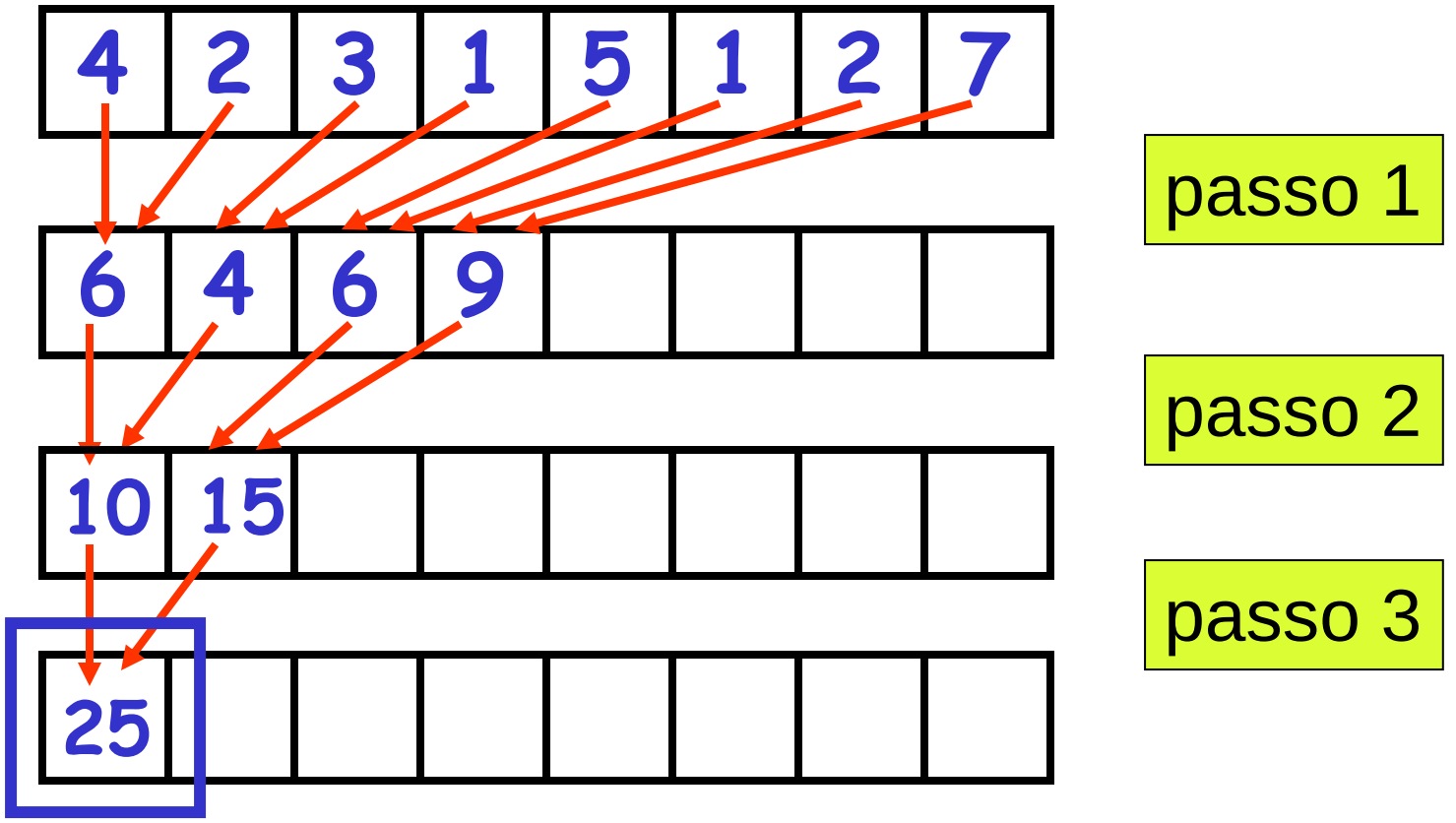

caso generale
n/2
n


$k=1,2,\ldots,\log_2 n$
livello 0
sono 2^k
livello k
n


porzione 1..n/2 dell'array
n/2
array 1D di size n
n

somma degli elementi di un array 1D di size n

algoritmo *in place*



dati di input: l'array (variabile **a**), il size dell'array (variabile **n**)

dato di output: la somma degli elementi (variabile **somma**)

costrutto ripetitivo: **for** ($\log_2 n$ passi)

operazione ripetuta (al generico passo **k**)**:**

calcolare le somme di tutte le coppie del livello **k**

ciclo **for** (al generico passo **i**)**:**

sommare l'**i**-sima coppia, ovvero l'elemento (2***i**)-simo e l'elemento (2***i**-1)-simo

al passo **k** (livello **k**) ci sono 2^k numeri da sommare, ovvero 2^{k-1} coppie

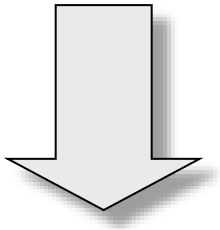
```
int somma_raddoppiamento(int a[], int
 int i, k, somma;
 for (k=log2(n); k> 0; k=k-1)
    for (i=1; i <= 2^(k-1)
        a[i-1] = a[2*(i
    }
 }
 somma =
 retu
}
```

TROPPO COMPLICATO !

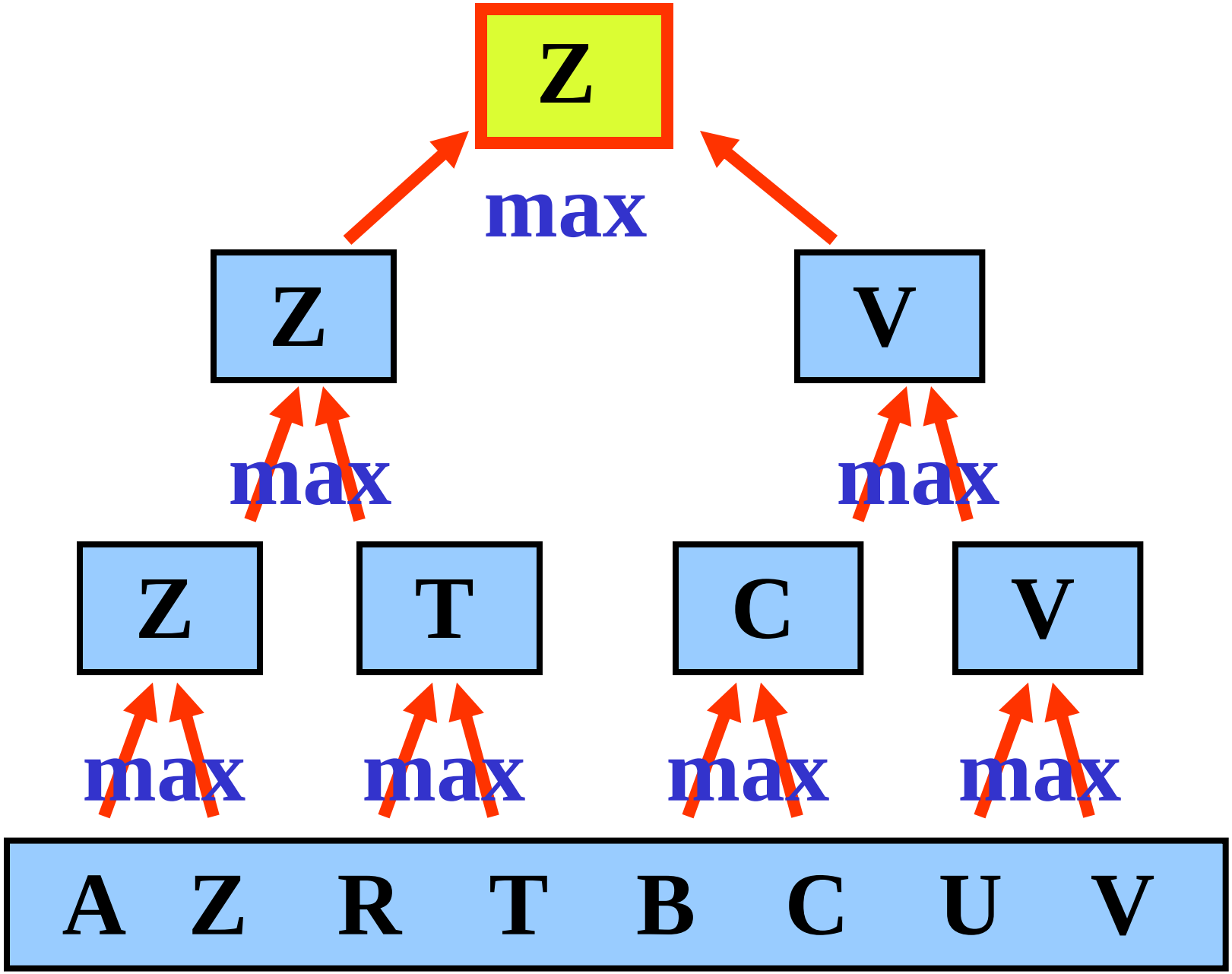
T(n)=n-1
somme
(tra elementi
dell'array)

stessa complessità dell'algoritmo incrementale

approccio **divide et impera** al problema della determinazione del **massimo elemento** di un array di size *n*



l'algoritmo ha lo stesso scheletro di controllo di **somma_raddoppiamento**


Z
max
Z
V
max
max
Z
T
C
V
max
max
max
max
A Z R T B C U V

```
int max_raddoppiamento(int a[], int n) {
 int i, k, massimo;
 for (k=log2(n); k> 0; k=k-1) {
    for (i=1; i <= 2^(k-1); i++) {
        a[i-1] = max(a[2*(i-1)], a[2*(i-1)+1]) ;
    }
 }
 massimo = a[0];
 return massimo;
}
```

TROPPO COMPLICATO !

T(n)=n-1 confronti (tra elementi dell'array)

```
int max (int x, int y) {
 if (x > y)
    { return x; }
 else
    { return y; }
}
```

stessa complessità dell'algoritmo incrementale

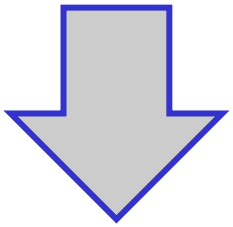
struttura di un algoritmo di raddoppiamento (divide et impera)

```
...
for (k=log2(n); k> 0; k=k
    for (i=1; i <= 2^(k
        azione sull
        elemen
    }
}
...
```

k-simo livello

TROPPO COMPLICATO !

esiste un modo *più semplice* per descrivere un algoritmo basato sull'approccio **divide et impera ?**



tecnica di
programmazione ricorsiva