

Programmazione I-B 2022-23

Laboratorio T2

(penultima cifra matricola PARI)

Attilio Fiandrotti

attilio.fiandrotti@unito.it

10 Novembre 2022

Outline

- Soluzione esercizi 3 Novembre
- Visualizziamo la ricorsione
- Esercizi sulla ricorsione

Soluzione esercizi 3 Novembre

Esercizio: serie numeriche

- Di seguito sono elencate serie numeriche ed il valore cui convergono

$$\sum_{k=0}^n q^k = \frac{1 - q^{n+1}}{1 - q} \quad (1)$$

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} \quad (2)$$

$$\sum_{k=0}^{\infty} \frac{1}{(2k+1)^2} = \frac{\pi^2}{8} \quad (3)$$

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = \frac{\pi}{4} \quad (4)$$

$$\sum_{k=0}^{\infty} \frac{z^k}{k!} = e^z \quad (5)$$

Esercizio: serie numeriche

- Di seguito sono elencate serie numeriche ed il valore cui convergono

$$\sum_{k=0}^n q^k = \frac{1 - q^{n+1}}{1 - q} \quad (1)$$

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} \quad (2)$$

$$\sum_{k=0}^{\infty} \frac{1}{(2k+1)^2} = \frac{\pi^2}{8} \quad (3)$$

Esponenziale

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = \frac{\pi}{4} \quad (4)$$

Fattoriale


$$\sum_{k=0}^{\infty} \frac{z^k}{k!} = e^z \quad (5)$$

Esercizio: serie 5 – V1

$$\sum_{k=0}^{\infty} \frac{z^k}{k!} = e^z$$

```
public static float quintaSerie(int z, int n) {  
    float s = 0;  
    int num = 1, den = 0;  
    int k = 0;  
    while (k < n + 1) {  
        num = Aritmetica.pot (z, k);  
        den = fattoriale(k);  
        s = s + ((float)num / (float)den);  
        k = k + 1;  
    }  
    return s;  
}
```

Termine k -esimo
della serie



```
public static int fattoriale(int n) {  
    int k = 1;  
    int ret = 1;  
    while (k <= n) {  
        ret = ret * k;  
        k = k + 1;  
    }  
    return ret;  
}
```

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

Esercizio: serie 5 – V2

- *potenza()* e *fattoriale()* invocate n volte

numero moltiplicazioni ordine n^2

- Riscriviamo *potenza(int z, int k)* ricordando che

$$z^k = z * z^{k-1}$$

- Riscriviamo *esponenziale(int n)* ricordando che

$$n! = n * (n-1)! = n * (n-1) * (n-2) * \dots * 1$$

Esercizio: serie 5 – V2

$$1 + \sum_{k=1}^{\infty} \frac{z^k}{k!} = e^z$$

```
public static float quintaSerieV2(int z, int n) {  
    float s = 1, termine = 0;  
    int num = 1, den = 1;  
    int k = 1;  
    while (k < n + 1) {  
        num = num * z;  
        den = den * k;  
        termine = (float)num / (float)den;  
        s = s + termine;  
        k = k + 1;  
    }  
    return s;  
}
```

Numero moltiplicazioni
cresce
solo **linearmente** con n

Combinazioni di carte

- Un giocatore riceve k carte da un mazzo di n carte. Si scriva un programma che richieda all'utente di inserire il numero di carte k ricevute e calcoli il numero di differenti combinazioni di k carte che può ricevere. Il programma verifichi che l'input inserito dall'utente sia corretto (es, $k < n$) e, se necessario, lo richieda nuovamente finché questo non è corretto.

Carte - Approccio

- Il numero di possibili combinazioni di k elementi estratti da un set di n elementi é dato dalla legge binomiale

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- L'operatore fattoriale $n!$ é calcolabile come

$$n! = \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

Carte - Approccio

- Si riutilizzino i metodi fattoriale() e binomiale() implementati nella classe Aritmetica.java
- Si sviluppi l'opportuna classe di test che si occupa di interagire con l'utente tramite i metodi della classe SIn ed esegua il calcolo richiesto
- Tutti risultati intermedi sono correttamente rappresentabili in formato intero con segno a 32 bit (int) ?

Carte – fattoriale e binomiale

```
public static int binomiale(int n, int k) {  
    return fattoriale(n) / (fattoriale(k) * fattoriale(n-k));  
}
```

```
public static int fattoriale(int n) {  
    int k = 1;  
    int ret = 1;  
    while (k <= n) {  
        ret = ret * k;  
        k = k + 1;  
    }  
    return ret;  
}
```

Carte – test case

```
public static void main(String[] args) {
    int n = 0, k = 0, nComb = 0;
    int nCarteMazzo = 40 // Carte nel mazzo

    boolean inputCorretto = false;
    while (inputCorretto == false) {
        inputCorretto = true;
        System.out.print("Inserire il numero n di carte nel mazzo: ");
        n = SIn.readInt(); // ipotesi: n inserito > 0
        if (n > nCarteMazzo) {
            System.out.println("ERRORE: n deve essere < " + nCarteMazzo);
            inputCorretto = false;
        }
    }

    // Parsing di k analogo (verificare che k < n)
    System.out.print("Inserire il numero k di carte estratte: ");
    k = SIn.readInt(); // ipotesi: k inserito > 0

    nComb = Aritmetica.binomiale(n, k);
    System.out.println("Estraendo " + k + " carte da un mazzo di "
        + n + " ci sono " + nComb + " combinazioni");
}
```

Carte – test case

```
> java CarteTest  
Inserire il numero n di carte nel mazzo: 4  
Inserire il numero k di carte estratte: 2  
Estraendo 2 carte da un mazzo di 4 ci sono 6 combinazioni
```

```
> java CarteTest  
Inserire il numero n di carte nel mazzo: 12  
Inserire il numero k di carte estratte: 1  
Estraendo 1 carte da un mazzo di 12 ci sono 12  
combinazioni
```

```
> java CarteTest  
Inserire il numero n di carte nel mazzo: 13  
Inserire il numero k di carte estratte: 1  
Estraendo 1 carte da un mazzo di 13 ci sono 4 combinazioni
```

```
public static int binomiale(int n, int k) {  
    return fattoriale(n) / (fattoriale(k) * fattoriale(n-k));  
}
```

Carte – debug di *fattoriale()*

- Il metodo *binomiale()* non ritorna il valore corretto
- Il metodo *binomiale()* utilizza *fattoriale()*

```
public static int binomiale(int n, int k) {  
    return fattoriale(n) / (fattoriale(k) * fattoriale(n-k));  
}
```

- Verifichiamo il funzionamento di *fattoriale()* per *n* «grandi»

```
public static int fattoriale(int n) {  
    int k = 1;  
    int ret = 1;  
    while (k <= n) {  
        ret = ret * k;  
        k = k + 1;  
    }  
    return ret;  
}
```

Carte – test case fattoriale

```
public class FattorialeTest {
    //Valori attesi dalla funzione fattoriale
    //https://it.wikipedia.org/wiki/Fattoriale
    // n          n!
    // ...
    //12    479'001'600 -> rappresentabile su 32 bit signed [-2^31, 2^31-1)
    //13   6'227'020'800 -> NON rappresentabile su 32 bit signed [-2^31, 2^31-1)
    public static void main(String[] args) {
        // Verifichiamo il funzionamento della funzione fattoriale per n = 13
        int n = [...];
        long outputAtteso = [...];
        // Qui memorizzeremo il risultato di fattoriale in un long
        long risultato = 0;

        /* Verifichiamo il funzionamento di fattoriale */
        long fatt = Aritmetica.fattoriale(n);
        System.out.println("fattoriale(" + n + ") = " + fatt +
                           " => " + (fatt == outputAtteso));
    }
}
```


Carte – test case fattoriale

```
> java FattorialeTest  
fattoriale(12) = 479001600 => true
```

```
> java FattorialeTest  
fattoriale(13) = 1932053504 => false
```

↑
Valore atteso 6 227 020 800 !

6 227 020 800 –

1 932 053 504 =

4 294 967 296

← Errore pari a 2^{32}

Carte – test case fattoriale

- Il massimo numero rappresentabile su *int* é 2 147 483 647
 - *Overflow* del risultato di 2^{32} bit
 - Il risultato di $13!$ non é rappresentabile su 32 bit, nemmeno se *unsigned*
- Occorrerà calcolare $13!$ su *long* in *fattoriale()*

Table 5.1 · Some basic types

Value type	Range
<i>Int</i>	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
<i>Long</i>	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)

Carte – test case fattoriale con *long*

```
/* Versione di fattoriale() che ritorna long */  
public static long fattorialeLong(int n) {  
    int k = 1;  
    long ret = 1;  
    while (k <= n) {  
        ret = k * ret;  
        k = k + 1;  
    }  
    return ret;  
}
```

```
> java FattorialeTest  
fattorialeLong(13) = 6227020800 => true
```

Visualizziamo la ricorsione

La ricorsione - ripasso

- Approccio *divide et impera*
 - Individuare un caso che si sa risolvere (*caso base*)
 - Ricondursi al caso base per gli altri casi
- Caveat
 - Out of memory (*stack overflow*)
 - Complessità chiamata a funzione

Ricorsione covariante crescente $[0, \dots, n)$

- Stampare i numeri naturali nel segmento $[0, \dots, n)$ con metodi ricorsivi covarianti - *StampaSegmCovFinale.java*
 - in ordine crescente - *stampaU()*
 - in ordine decrescente - *stampaD()*

Ricorsione covariante crescente $[0, \dots, n)$

- Esempio: stampiamo i numeri naturali $[0, \dots, n)$ in ordine crescente con un algoritmo covariante per il caso $n=3$

```
public class StampaSegmCovFinaleTest {  
  
    public static void main(String[] args){  
        System.out.println("--- Test stampaU");  
  
        StampaSegmCovFinale.stampaU(3);System.out.println();  
    }  
}
```

↑
stampaU() chiamato
prima volta per $n = 3$

Ricorsione covariante crescente [0, ..., n)

```
public static void stampaU(int n) {  
    if (n == 0) {  
        /* blocco di codice vuoto */  
    } else {  
        stampaU(n-1);  
        System.out.print(n-1 + " ");  
    }  
    return;  
}
```


Ricorsione covariante crescente [0, ..., n)

```
public static void stampaU(int n) {  
    if (n == 0) {  
        /* blocco di codice vuoto */  
    } else {  
        stampaU(n-1);  
        System.out.print(n-1 + " ");  
    }  
    return;  
}
```

Caso generale $n \neq 0$
Stampa da 0 a $n-1$

Ricorsione covariante crescente [0, ..., n)

```
public static void stampaU(int n) {  
    if (n == 0) {  
        /* blocco di codice vuoto */  
    } else {  
        stampaU(n-1);  
        System.out.print(n-1 + " ");  
    }  
    return;  
}
```

Caso base $n==0$
stampa nulla

Caso generale $n \neq 0$
Stampa da 0 a $n-1$

Ricorsione covariante crescente $[0, \dots, n)$

```
public static void stampaU(int n) {  
    if (n == 0) {  
        /* blocco di codice vuoto */  
    } else {  
        stampaU(n-1);  
        System.out.print(n-1 + " ");  
    }  
    return;  
}
```

Caso base $n==0$
stampa nulla

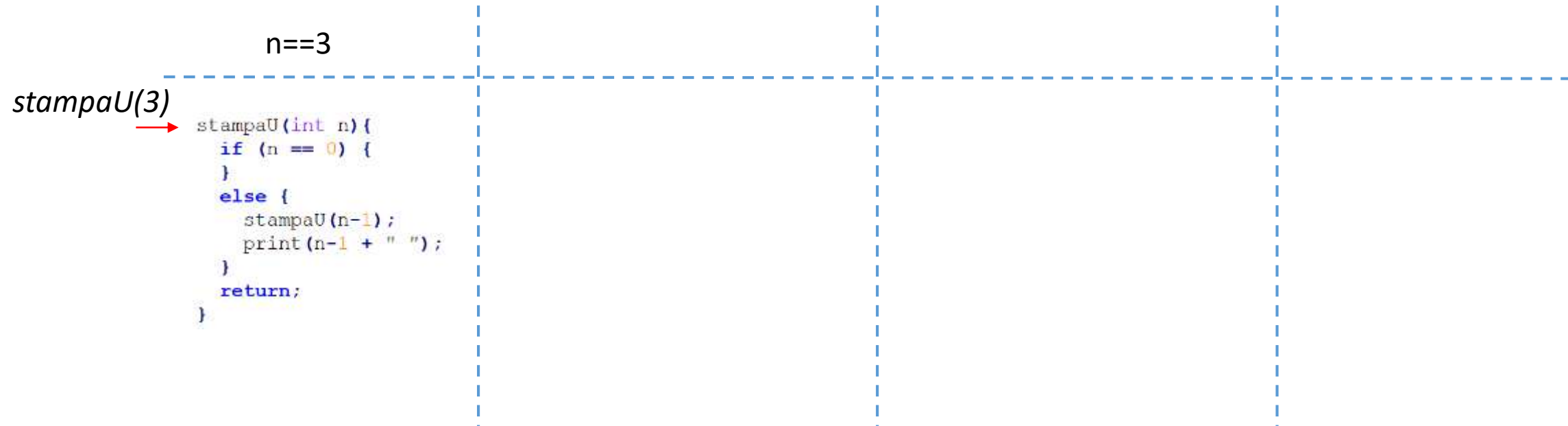
Caso generale $n \neq 0$
Stampa da 0 a $n-1$

return esplicitato
per fini didattici

Ricorsione covariante crescente $[0, \dots, n)$

```
>java StampaSegmCovFinaleTest
```

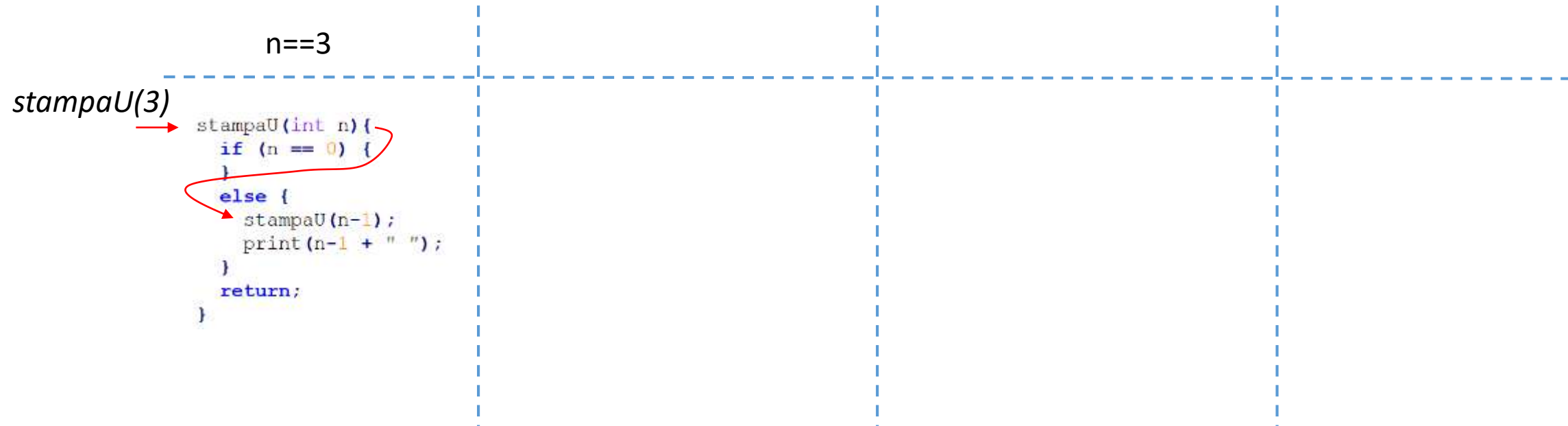
- Esempio per $n==3$



Ricorsione covariante crescente $[0, \dots, n)$

```
>java StampaSegmCovFinaleTest
```

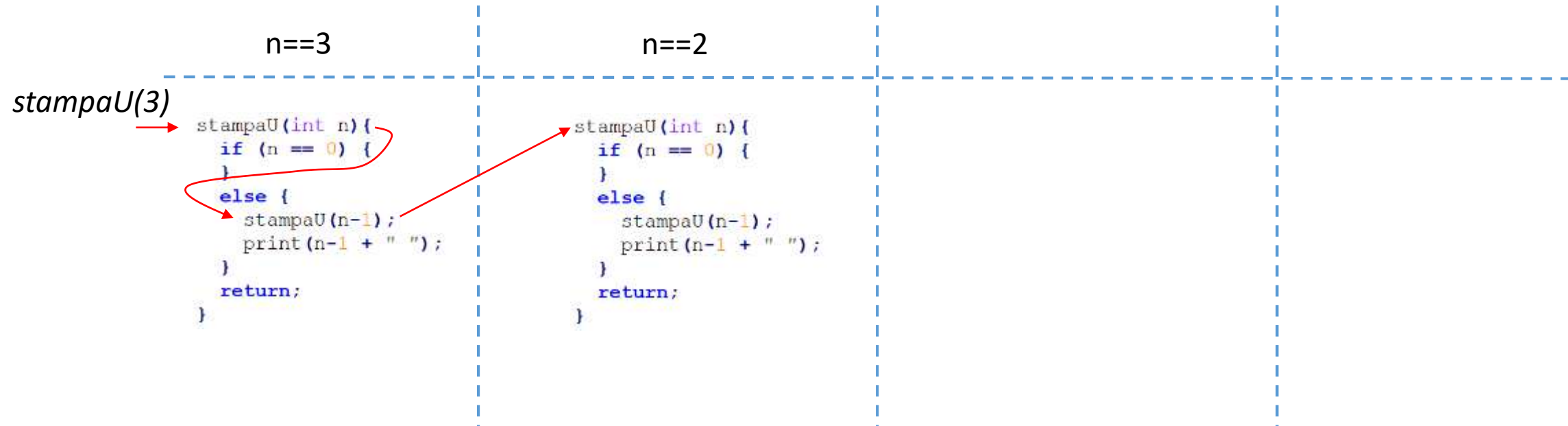
- Esempio per $n==3$



Ricorsione covariante crescente $[0, \dots, n)$

```
>java StampaSegmCovFinaleTest
```

- Esempio per $n==3$

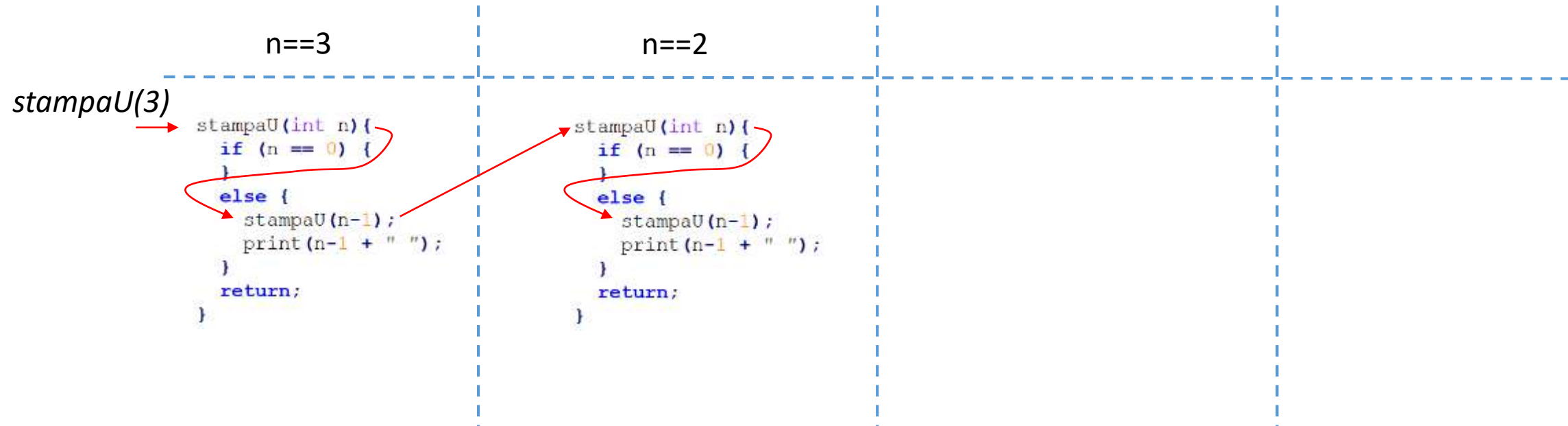


Chiamata a `stampaU(2)`

Ricorsione covariante crescente $[0, \dots, n)$

```
>java StampaSegmCovFinaleTest
```

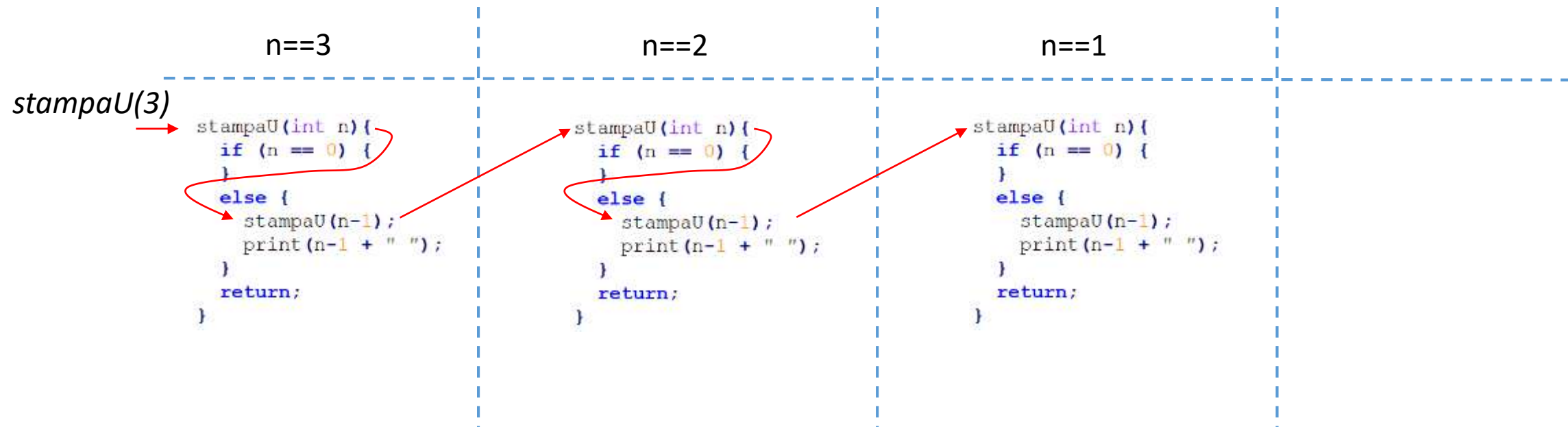
- Esempio per $n==3$



Ricorsione covariante crescente $[0, \dots, n)$

```
>java StampaSegmCovFinaleTest
```

- Esempio per $n==3$

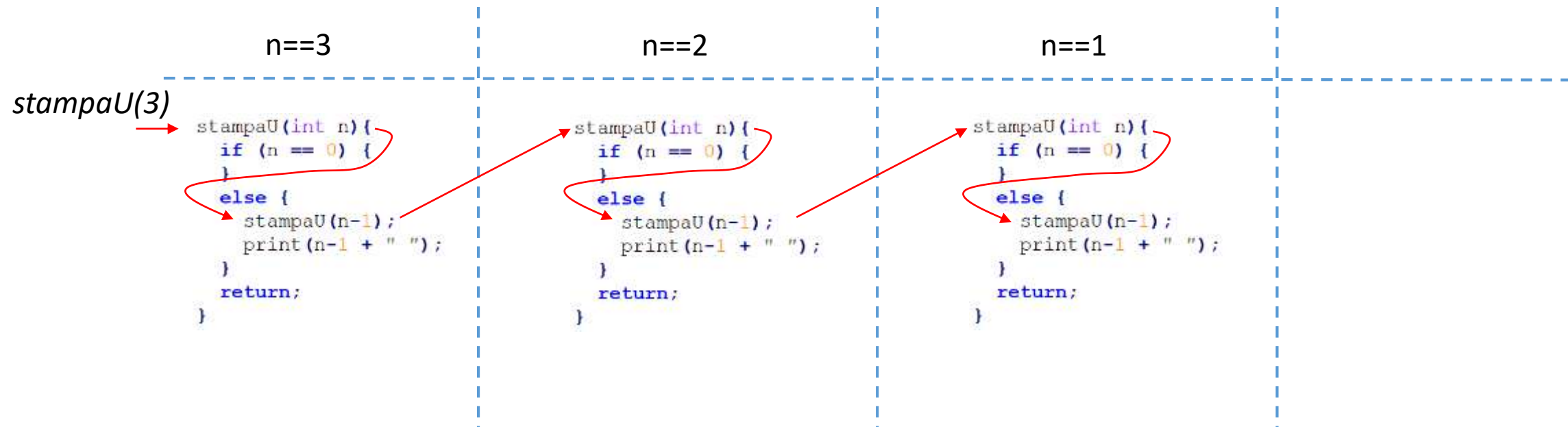


Chiamata a `stampaU(1)`

Ricorsione covariante crescente $[0, \dots, n)$

```
>java StampaSegmCovFinaleTest
```

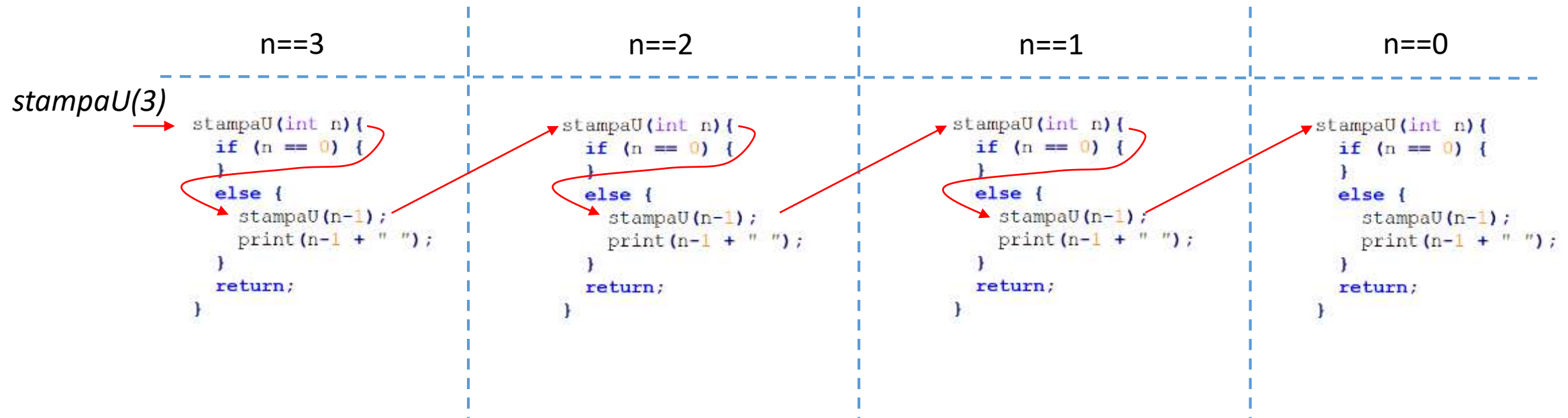
- Esempio per $n==3$



Ricorsione covariante crescente [0, ..., n)

```
>java StampaSegmCovFinaleTest
```

- Esempio per $n==3$

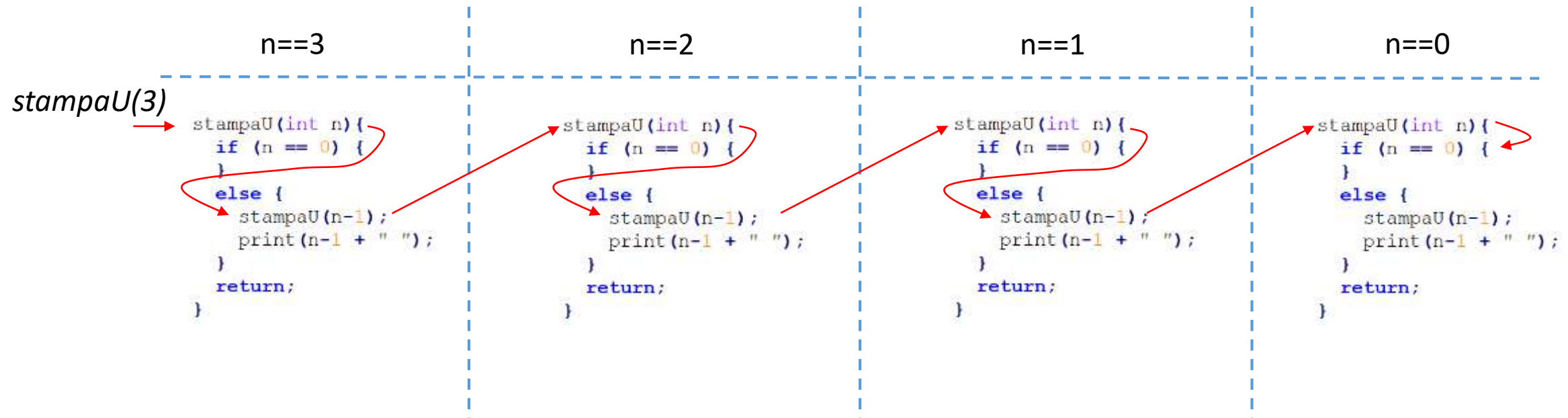


Chiamata a `stampaU(0)`

Ricorsione covariante crescente $[0, \dots, n)$

```
>java StampaSegmCovFinaleTest
```

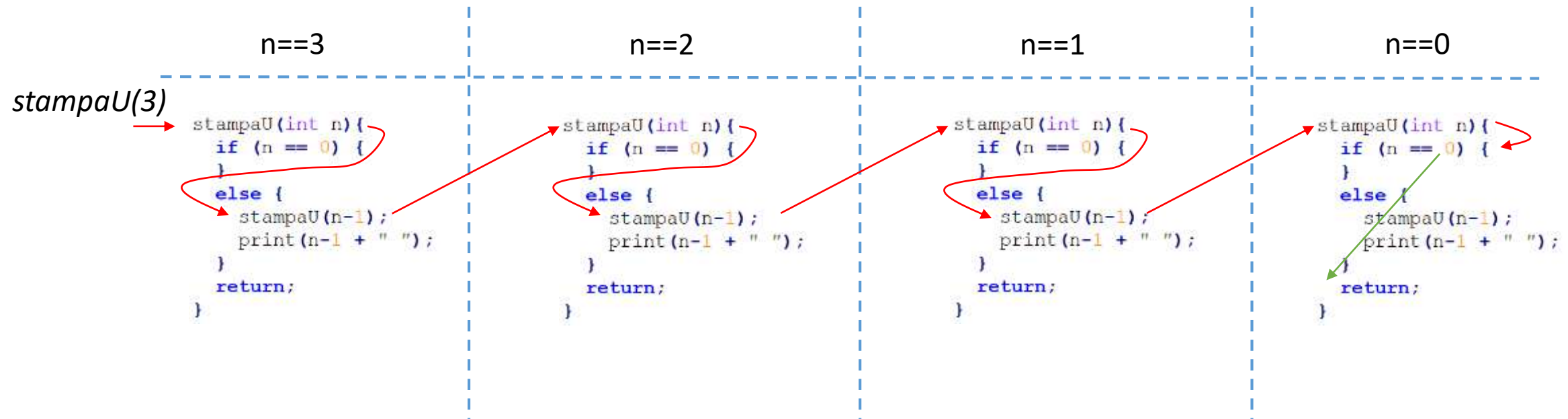
- Esempio per $n==3$



Ricorsione covariante crescente $[0, \dots, n)$

```
>java StampaSegmCovFinaleTest
```

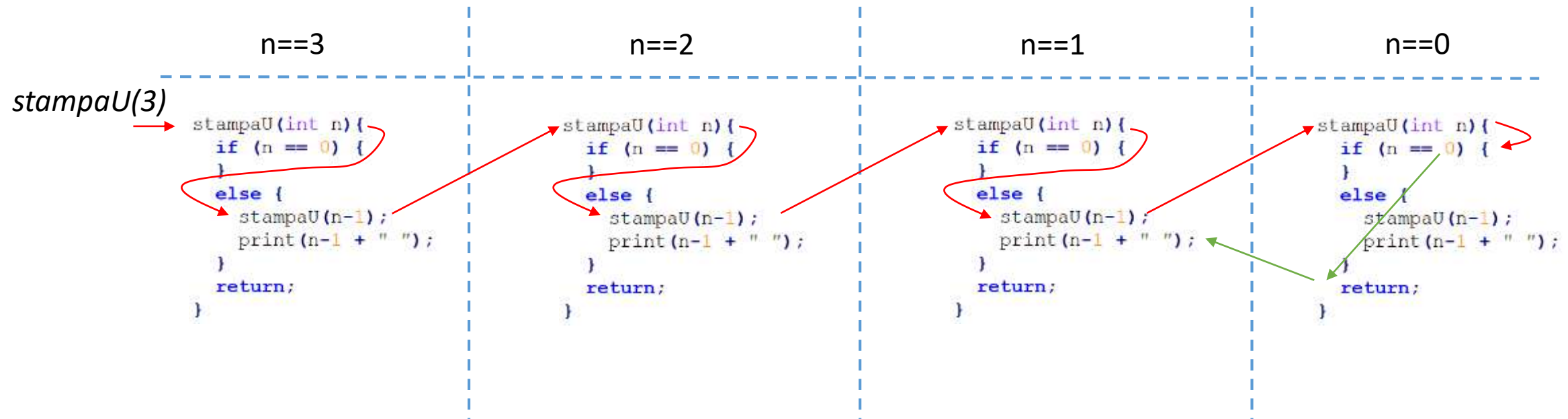
- Esempio per $n==3$



Ricorsione covariante crescente [0, ..., n)

```
>java StampaSegmCovFinaleTest
```

- Esempio per $n==3$

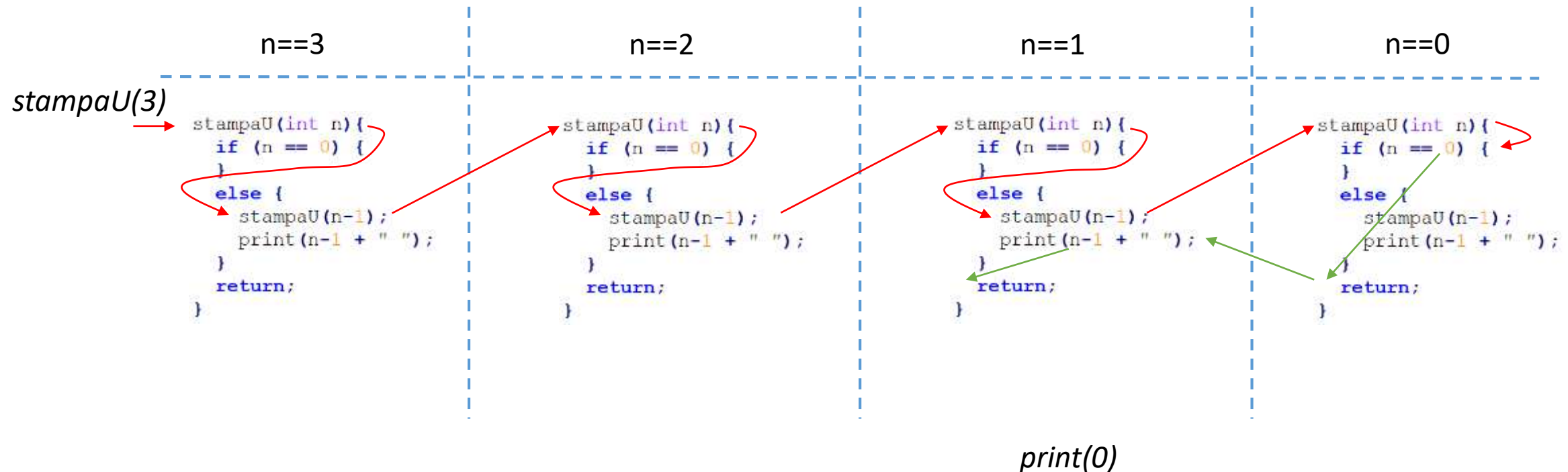


Ritorno a `stampaU(1)`

Ricorsione covariante crescente [0, ..., n)

- Esempio per $n=3$

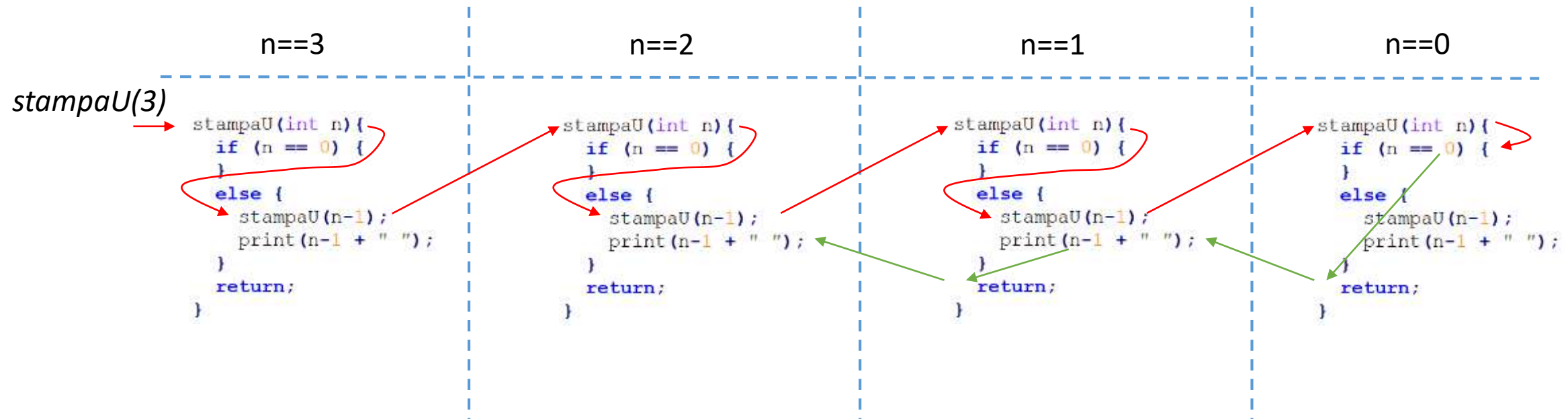
```
>java StampaSegmCovFinaleTest  
0
```



Ricorsione covariante crescente [0, ..., n)

- Esempio per $n=3$

```
>java StampaSegmCovFinaleTest  
0
```

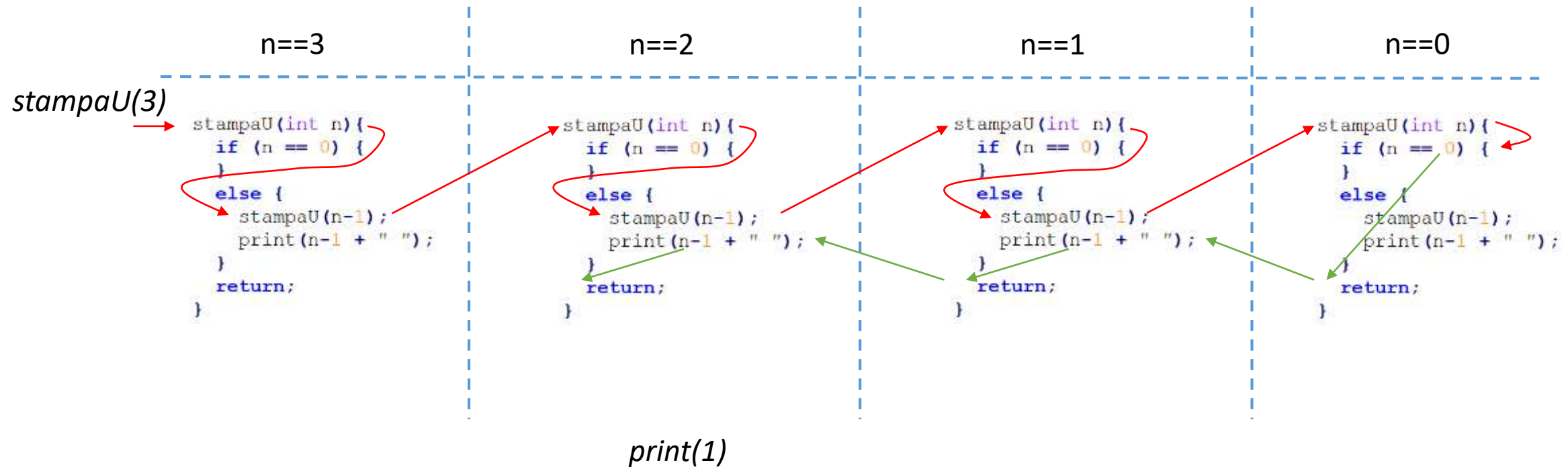


Ritorno a `stampaU(2)`

Ricorsione covariante crescente [0, ..., n)

- Esempio per $n=3$

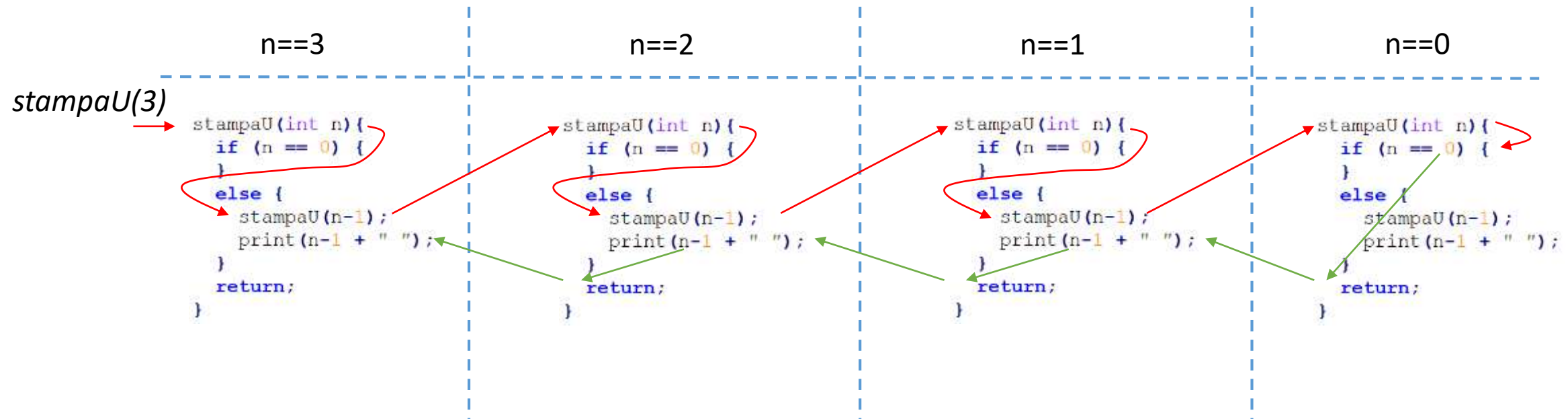
```
>java StampaSegmCovFinaleTest  
0 1
```



Ricorsione covariante crescente $[0, \dots, n)$

- Esempio per $n==3$

```
>java StampaSegmCovFinaleTest  
0 1
```

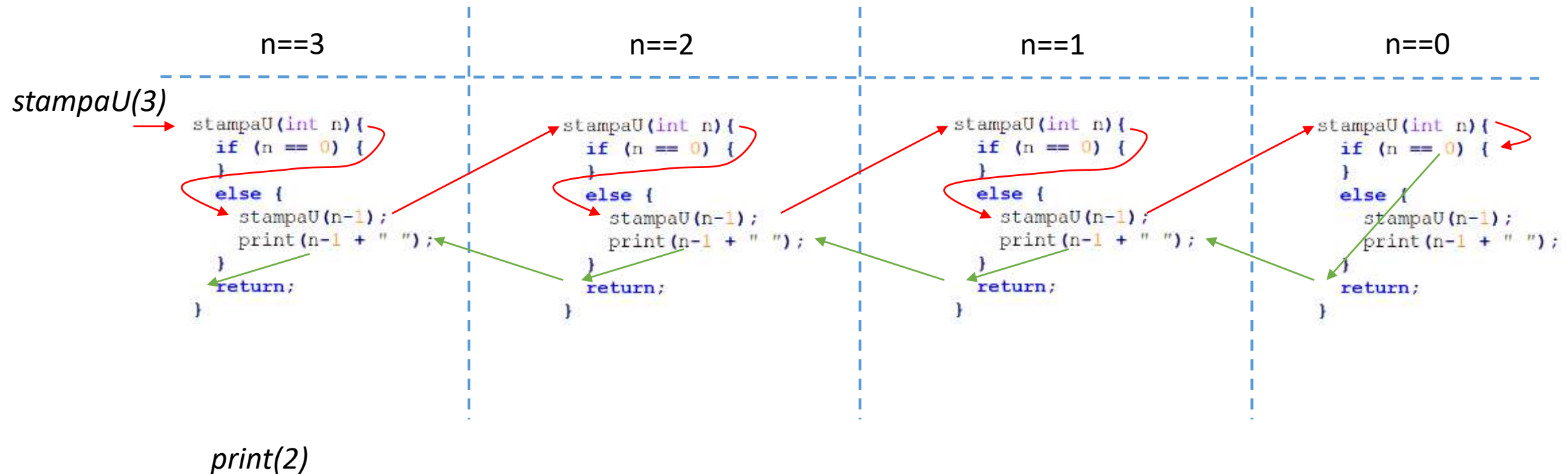


Ritorno a `stampaU(3)`

Ricorsione covariante crescente [0, ..., n)

- Esempio per $n==3$

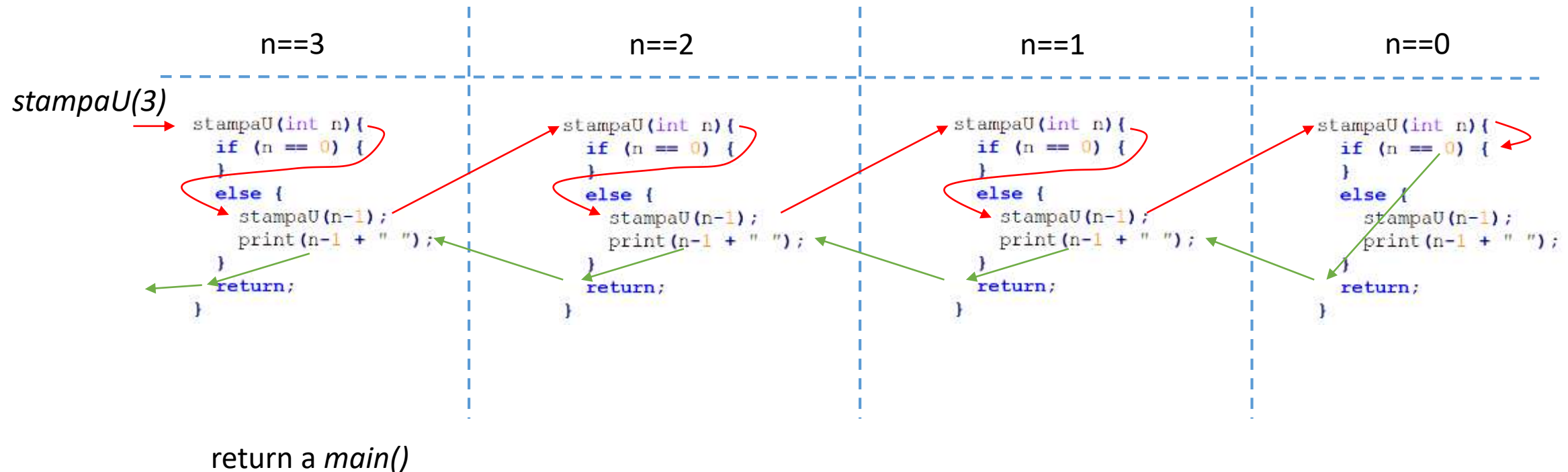
```
>java StampaSegmCovFinaleTest  
0 1
```



Ricorsione covariante crescente $[0, \dots, n)$

- Esempio per $n==3$

```
>java StampaSegmCovFinaleTest  
0 1 2
```



Ricorsione covariante decrescente [0, n)

- Esempio: stampiamo i numeri naturali [0, ..., n) in ordine decrescente con un algoritmo covariante per il caso $n=3$

```
public class StampaSegmCovFinaleTest {  
  
    public static void main(String[] args){  
        System.out.println("--- Test stampaU");  
  
        StampaSegmCovFinale.stampaD(3);System.out.println();  
    }  
}
```

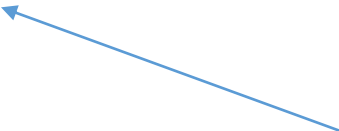
stampaD() chiamato
prima volta per $n = 3$

Ricorsione covariante decrescente [0, n)

```
public static void stampaD(int n) {  
    if (n == 0) {  
        /* blocco codice vuoto */  
    } else {  
        System.out.print(n-1 + " ");  
        stampaD(n-1);  
    }  
    return;  
}
```

Invertito!

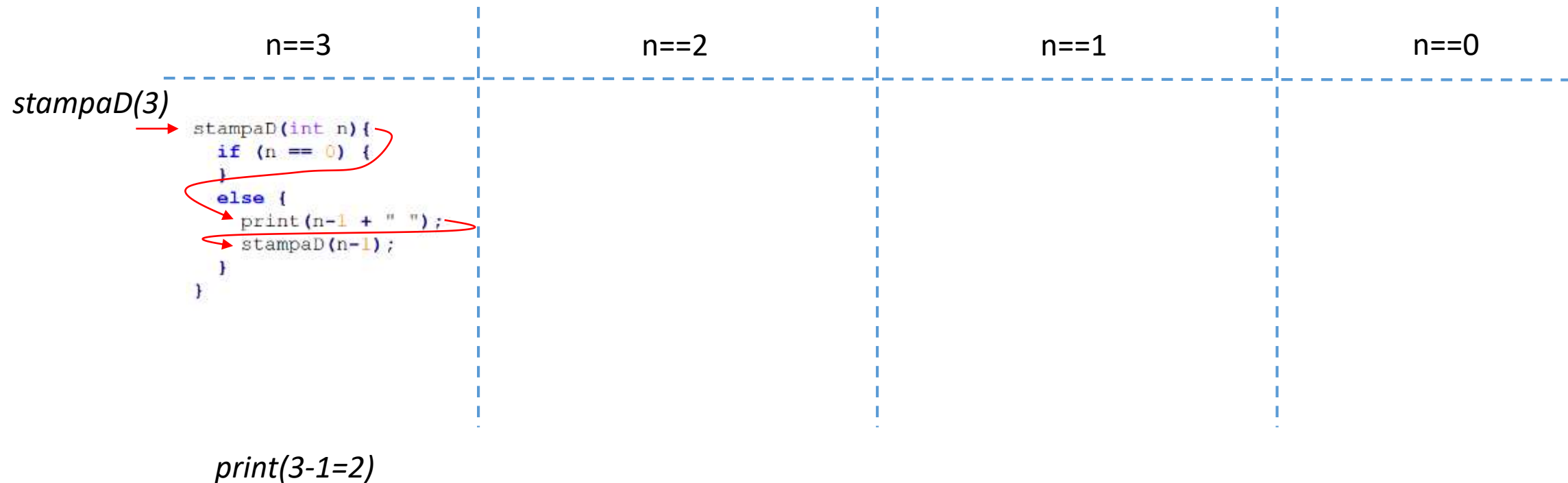
return omissso
in seguito



Ricorsione covariante decrescente [0, n)

- Esempio per n==3

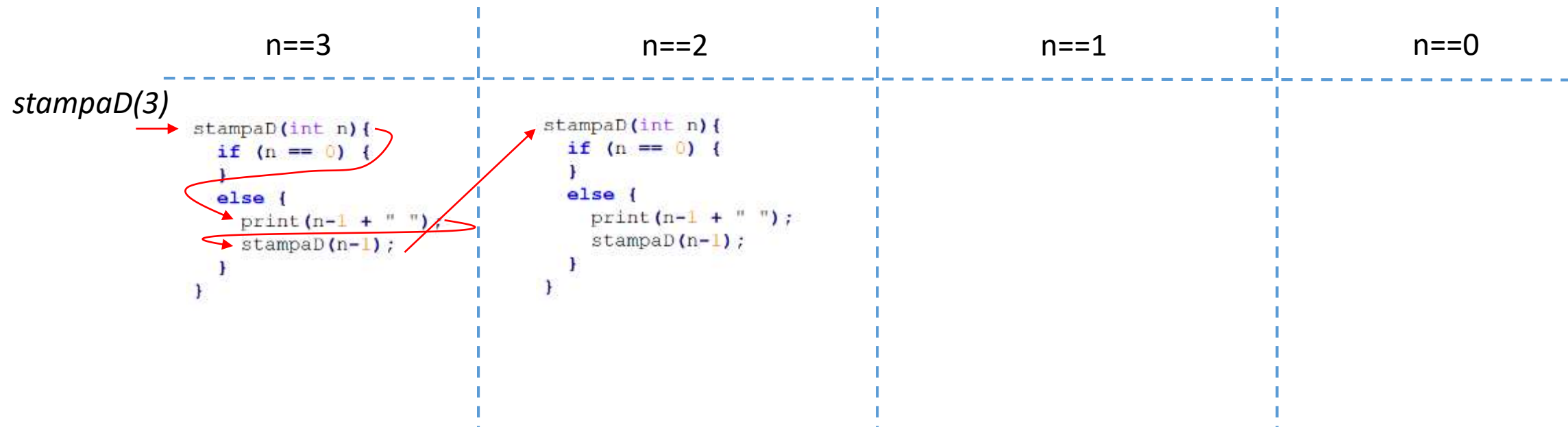
```
>java StampaSegmCovFinaleTest  
2
```



Ricorsione covariante decrescente $[0, n)$

- Esempio per $n==3$

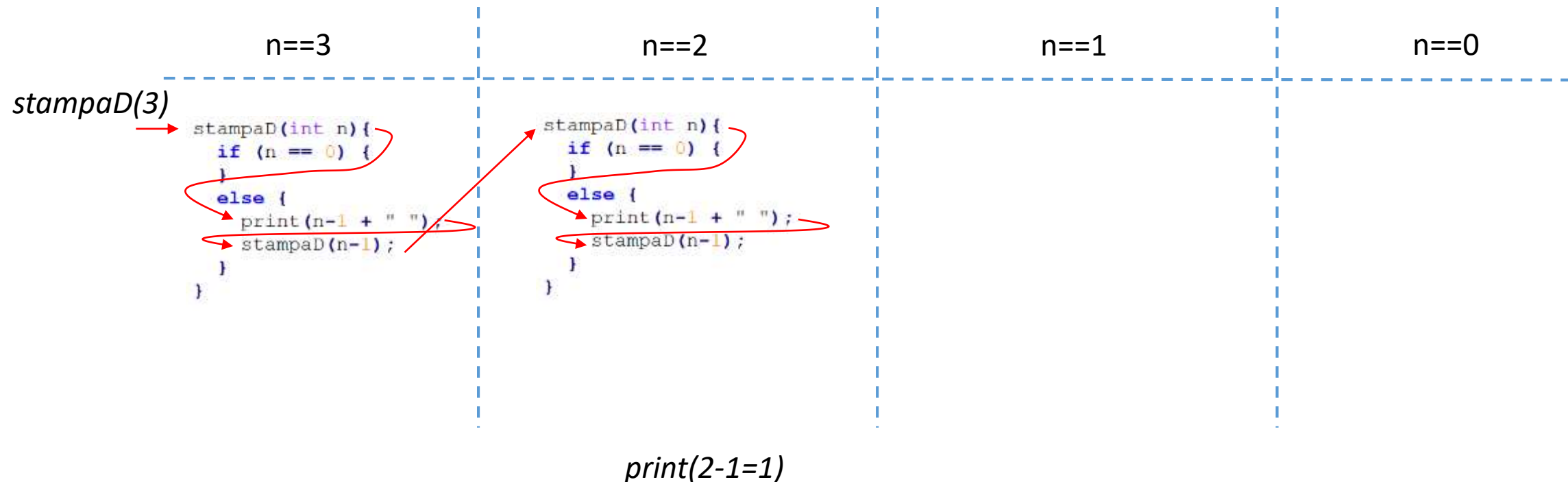
```
>java StampaSegmCovFinaleTest  
2
```



Ricorsione covariante decrescente $[0, n)$

- Esempio per $n==3$

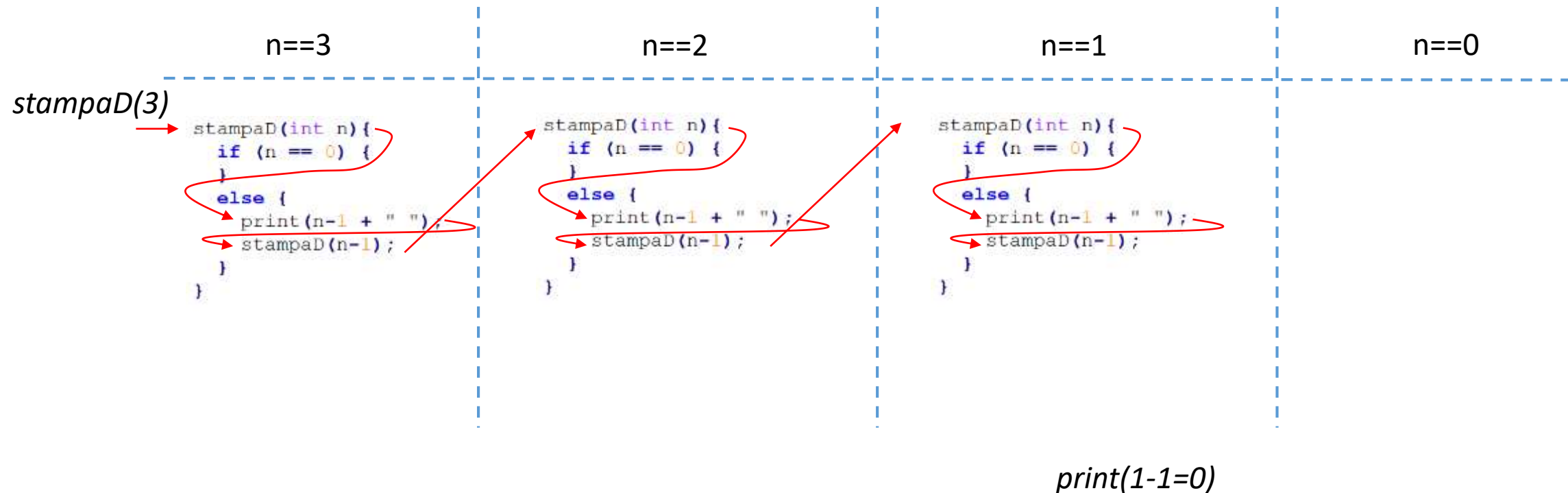
```
>java StampaSegmCovFinaleTest  
2 1
```



Ricorsione covariante decrescente [0, ..., n)

- Esempio per $n=3$

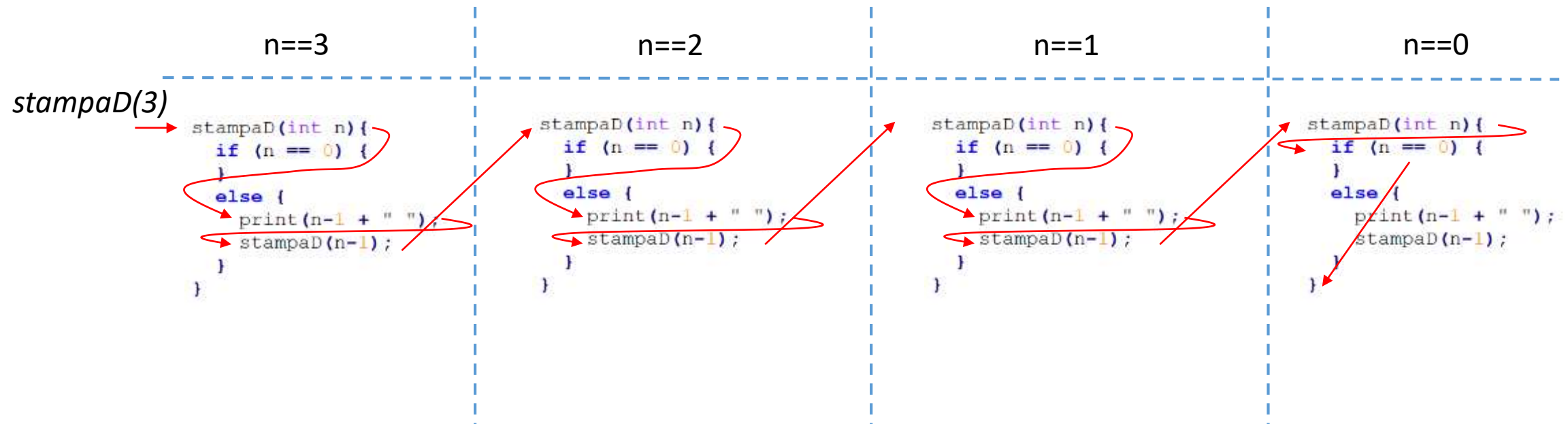
```
>java StampaSegmCovFinaleTest  
2 1 0
```



Ricorsione covariante decrescente [0, ..., n)

- Esempio per $n=3$

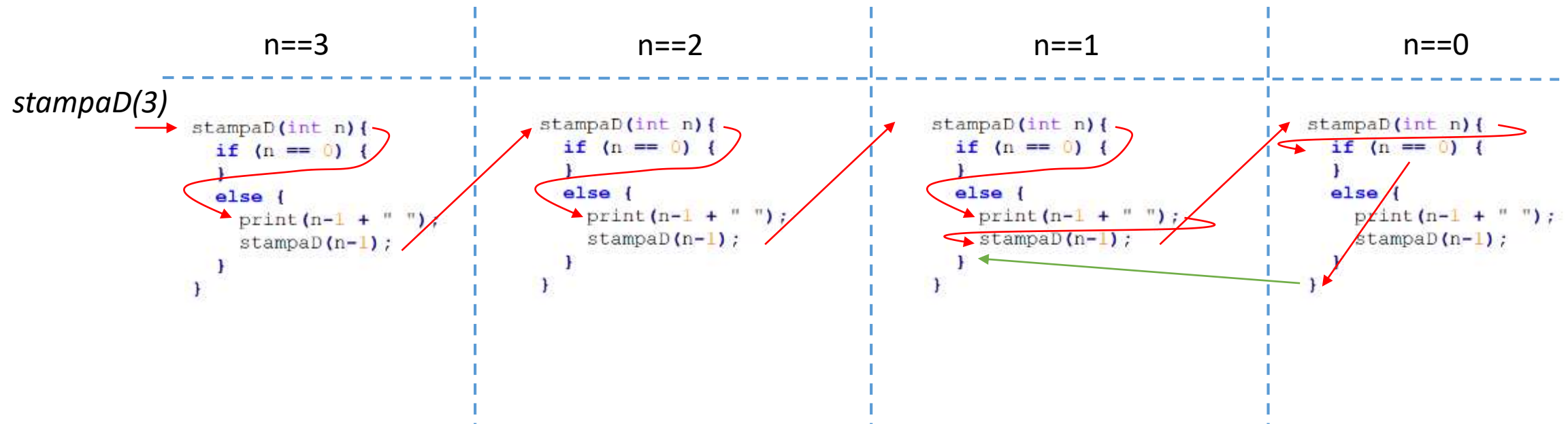
```
>java StampaSegmCovFinaleTest  
2 1 0
```



Ricorsione covariante decrescente [0, ..., n)

- Esempio per $n=3$

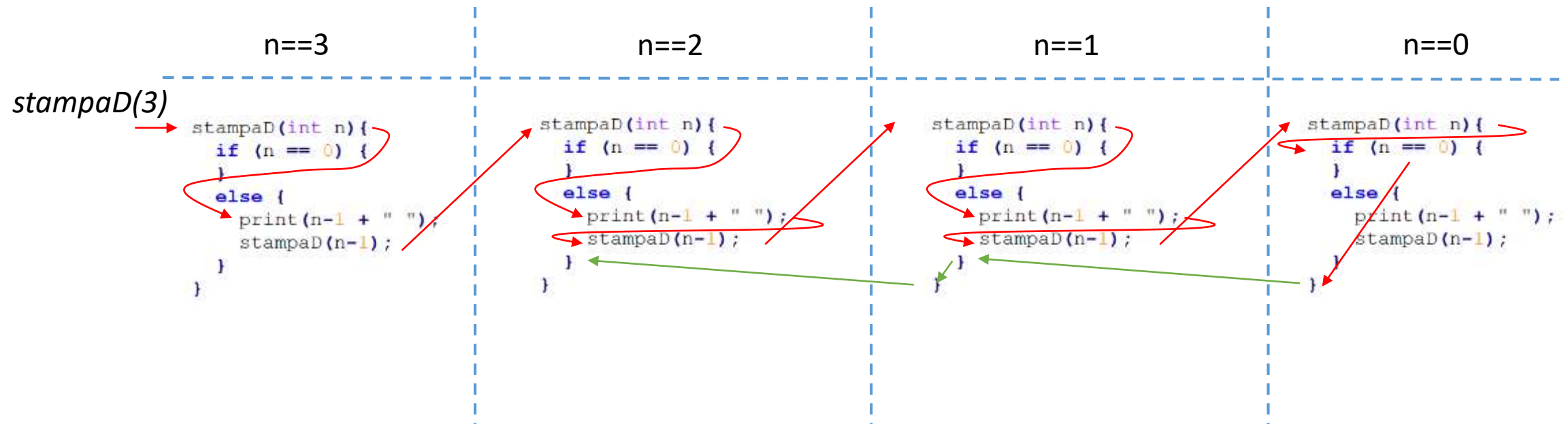
```
>java StampaSegmCovFinaleTest  
2 1 0
```



Ricorsione covariante decrescente [0, ..., n)

- Esempio per $n=3$

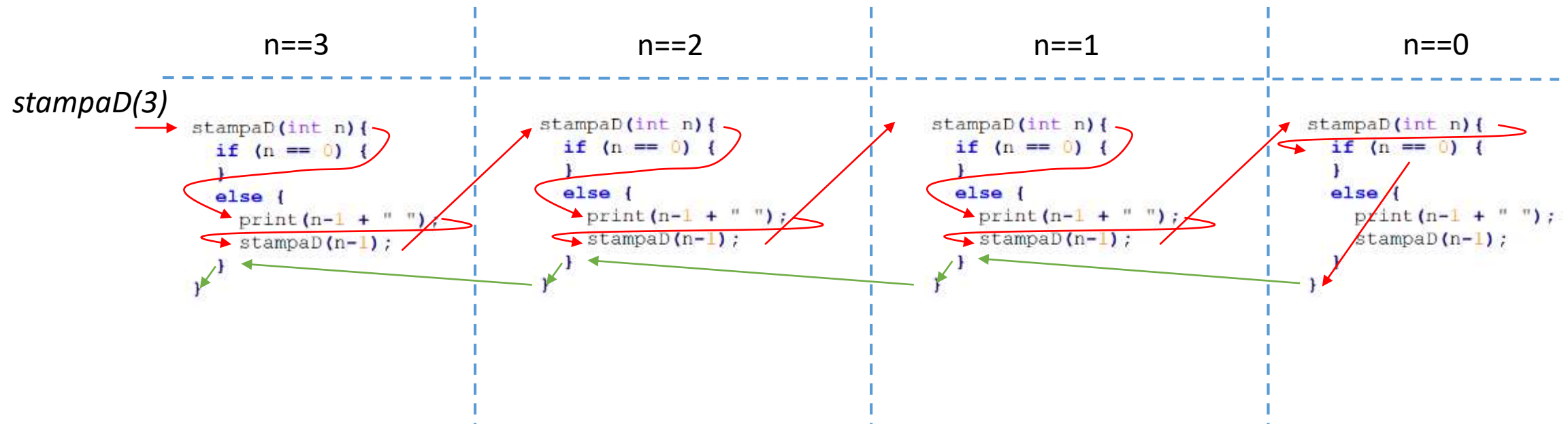
```
>java StampaSegmCovFinaleTest  
2 1 0
```



Ricorsione covariante decrescente [0, ..., n)

- Esempio per $n=3$

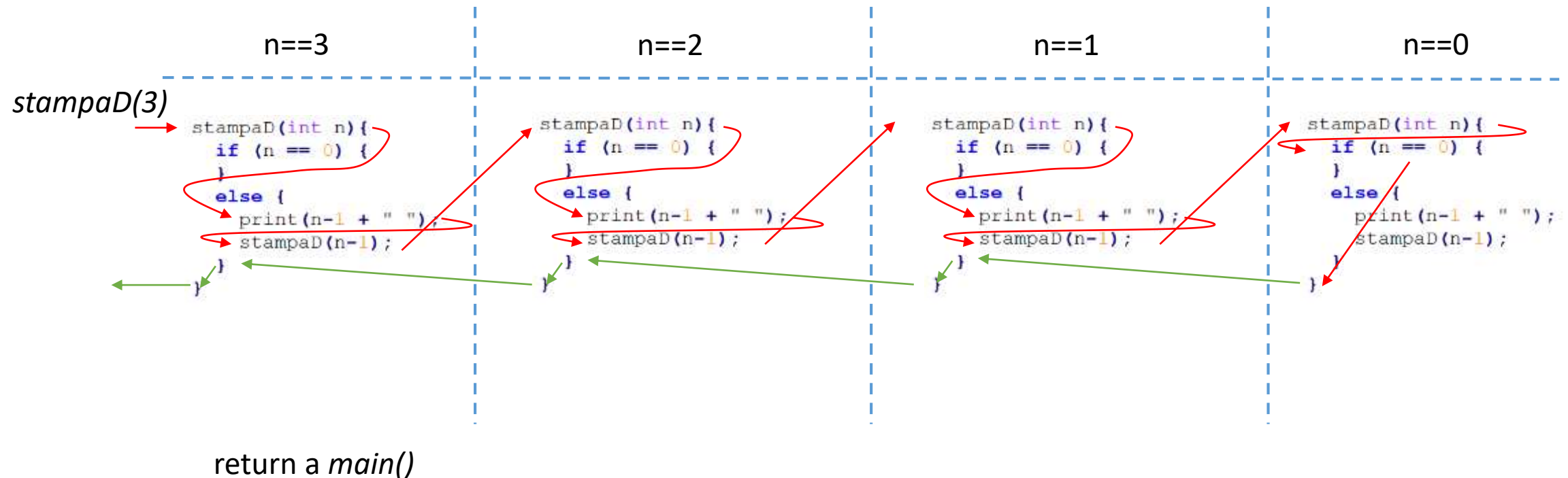
```
>java StampaSegmCovFinaleTest  
2 1 0
```



Ricorsione covariante decrescente [0, ..., n)

- Esempio per $n=3$

```
>java StampaSegmCovFinaleTest  
2 1 0
```



Ricorsione controvariante crescente $[0, \dots, n)$

- Esempio: stampiamo i numeri naturali $[0, \dots, n)$ in ordine crescente con un algoritmo controvariante per il caso $n=3$

```
public class StampaSegmConFinaleTest {  
  
    public static void main(String[] args) {  
        System.out.println("--- Test stampaU");  
  
        StampaSegmConFinale.stampaU(3); System.out.println();  
    }  
}
```

↑
stampaU() chiamato
prima volta per $n = 3$

Ricorsione controvariante crescente $[0, \dots, n)$

```
/* Wrapper del metodo controvariante */
public static void stampaU(int n){
    stampaU(n,0);
    return;
}

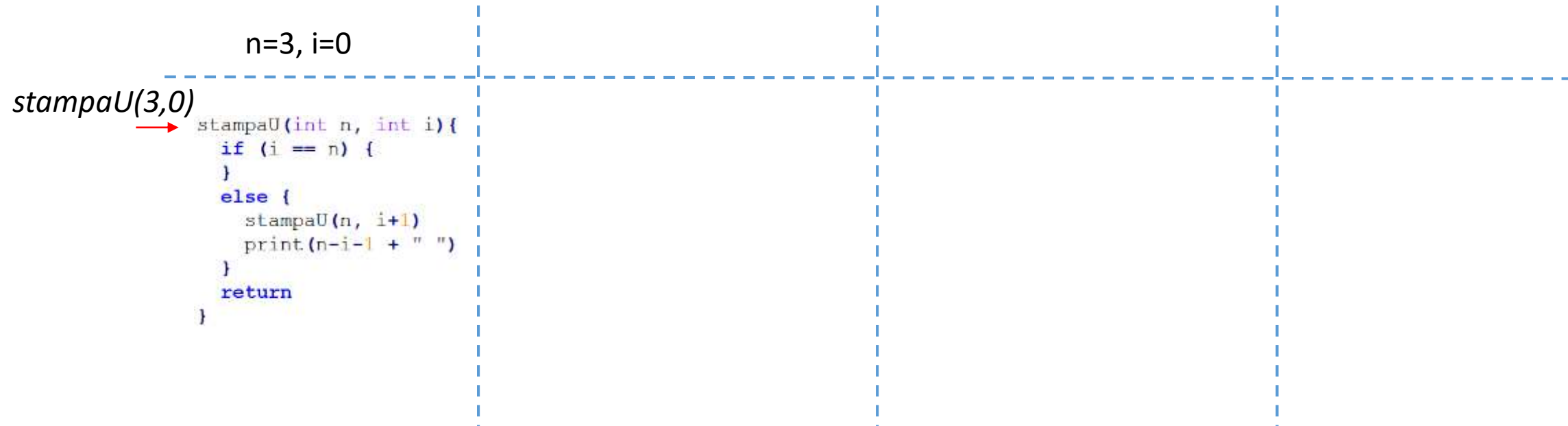
/* Metodo controvariante che stampa [0, n) */
public static void stampaU(int n, int i){
    if (n == i) {
        /* Stampa [0,n-n), cioè in [0,0), che è vuoto */
    } else {
        stampaU(n, i+1); /* Stampa [0,n-(i+1)), cioè [0,n-i-1) */
        System.out.print(n-(i+1) + " "); /* Stampa n-i-1*/
    }
    return;
}
```

Overloading di *stampaU()*

Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

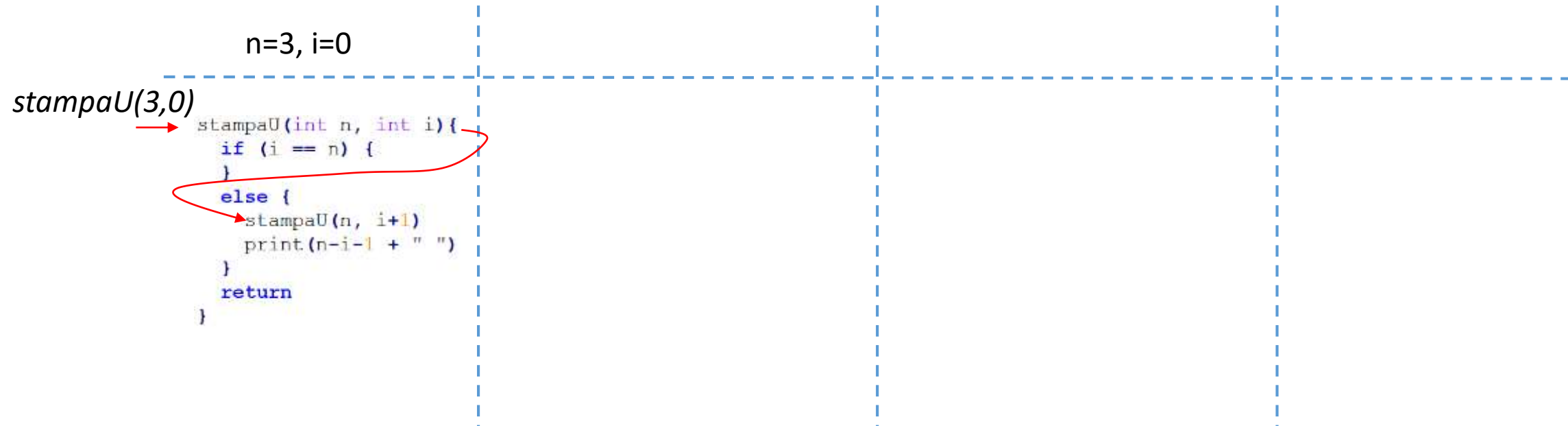
- Esempio per $n=3$



Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

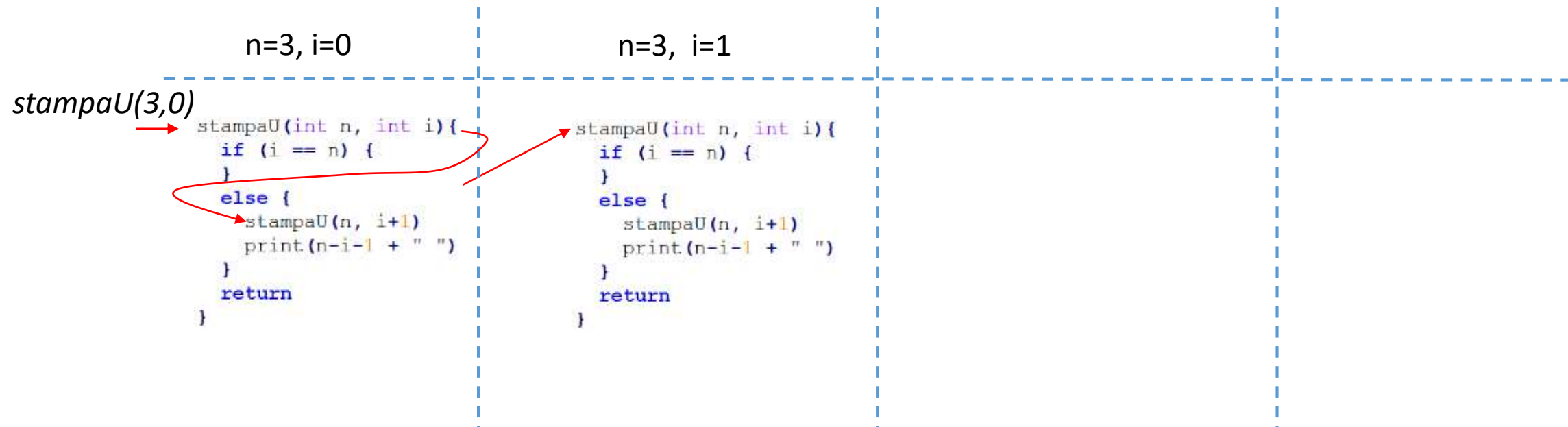
- Esempio per $n=3$



Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

- Esempio per $n=3$

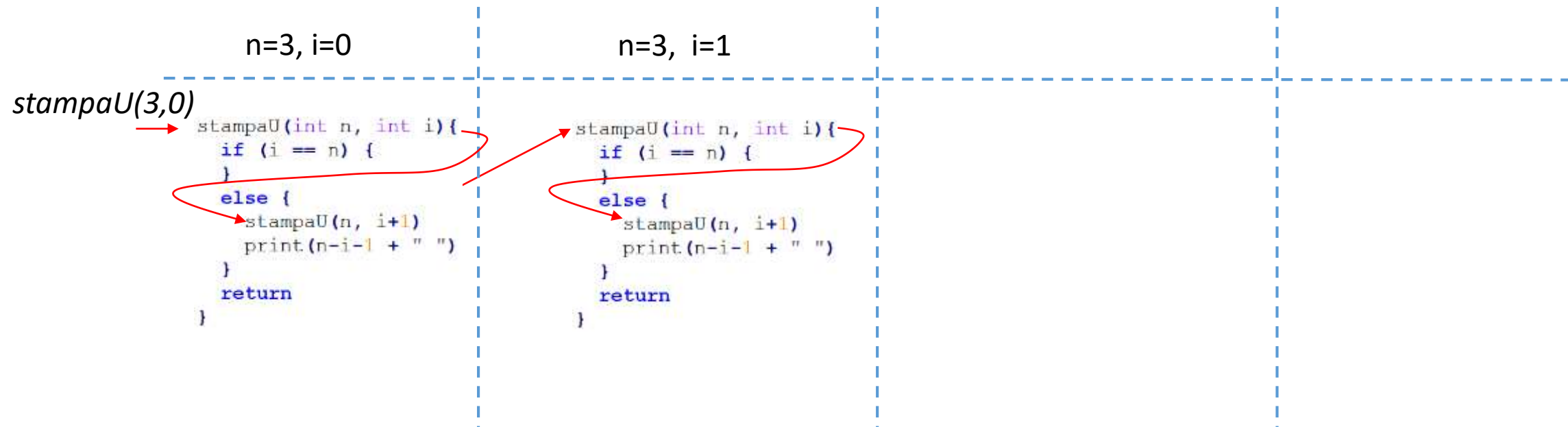


Chiamata a `stampaU(3,1)`

Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

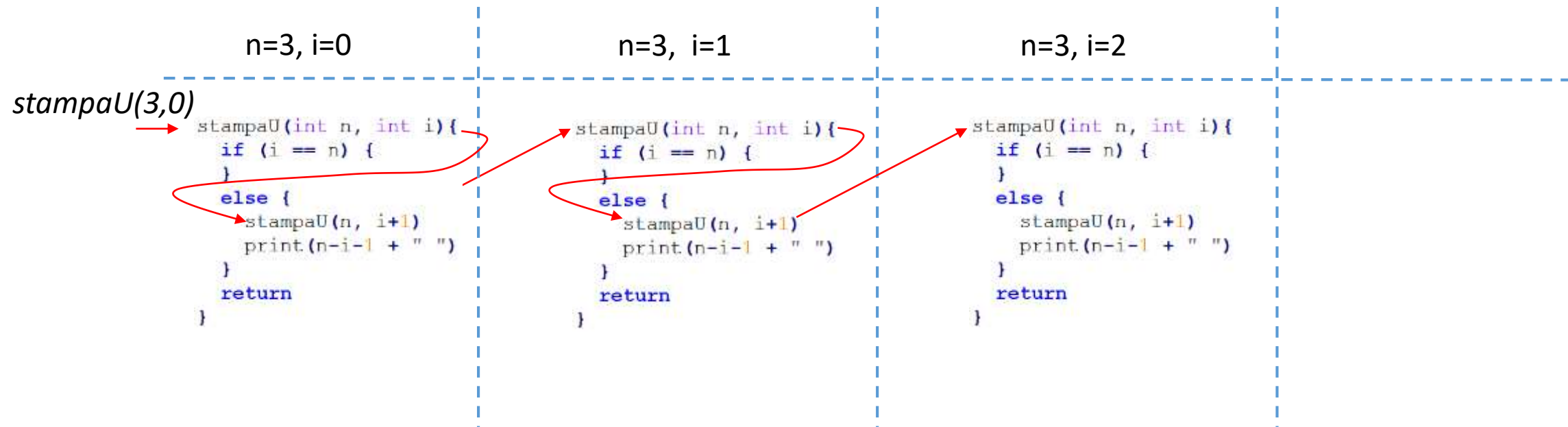
- Esempio per $n=3$



Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

- Esempio per $n=3$

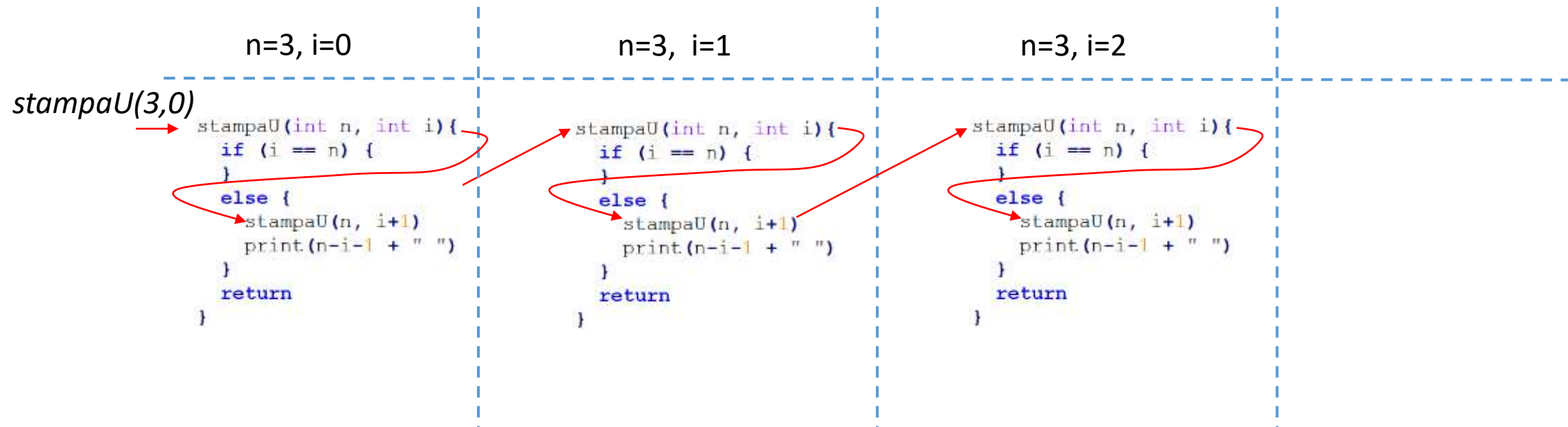


Chiamata a `stampaU(3,2)`

Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

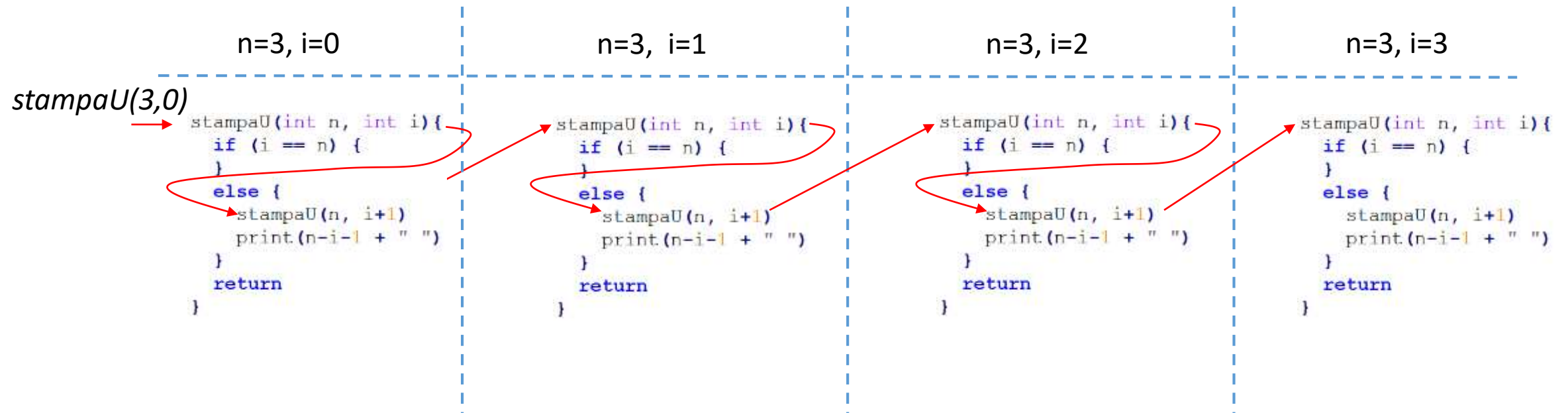
- Esempio per $n=3$



Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

- Esempio per $n=3$

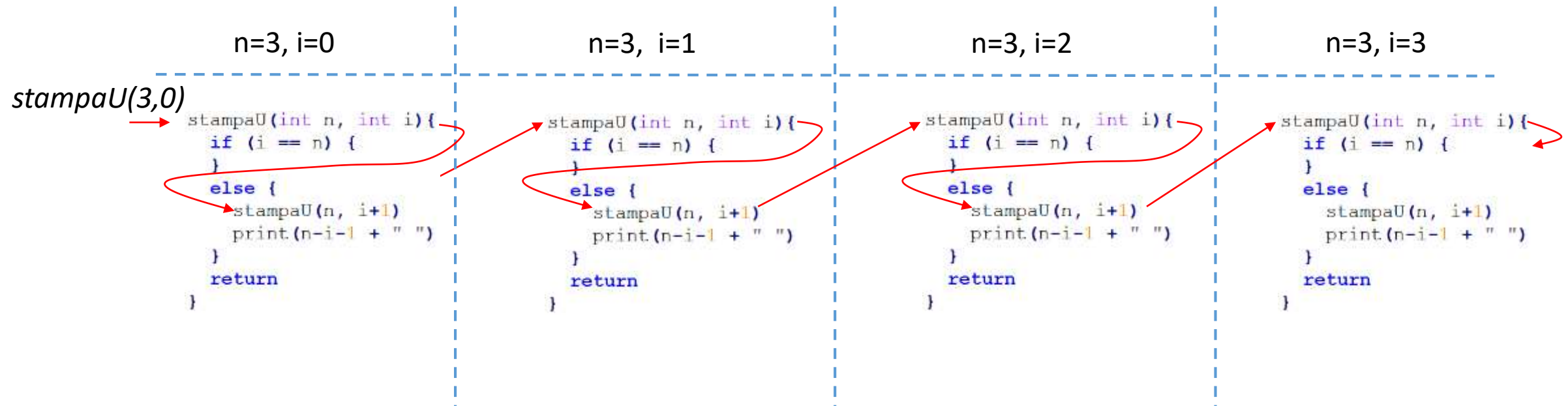


Chiamata a `stampaU(3,3)`

Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

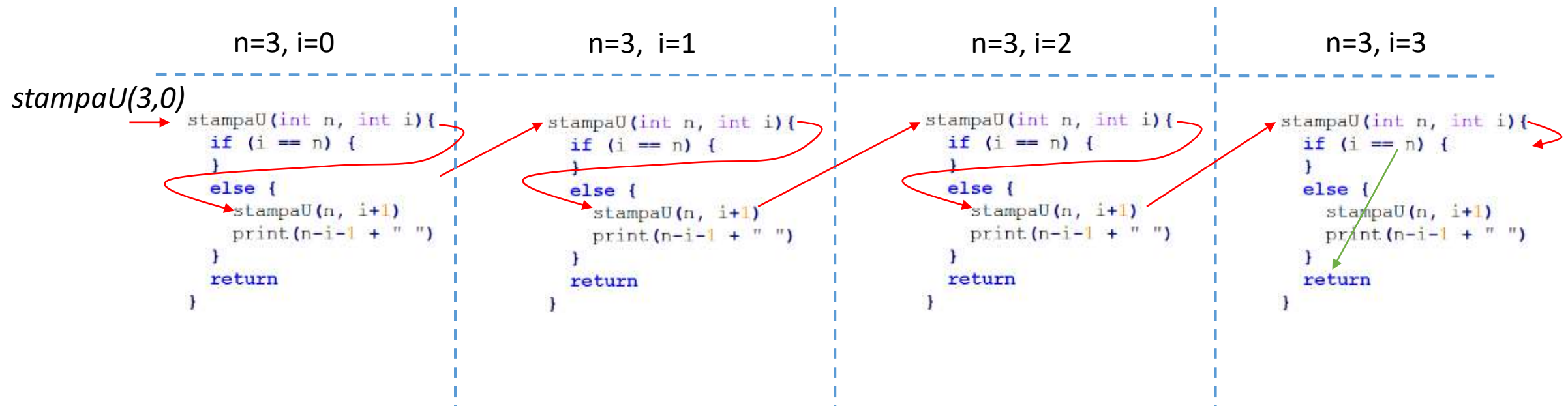
- Esempio per $n=3$



Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

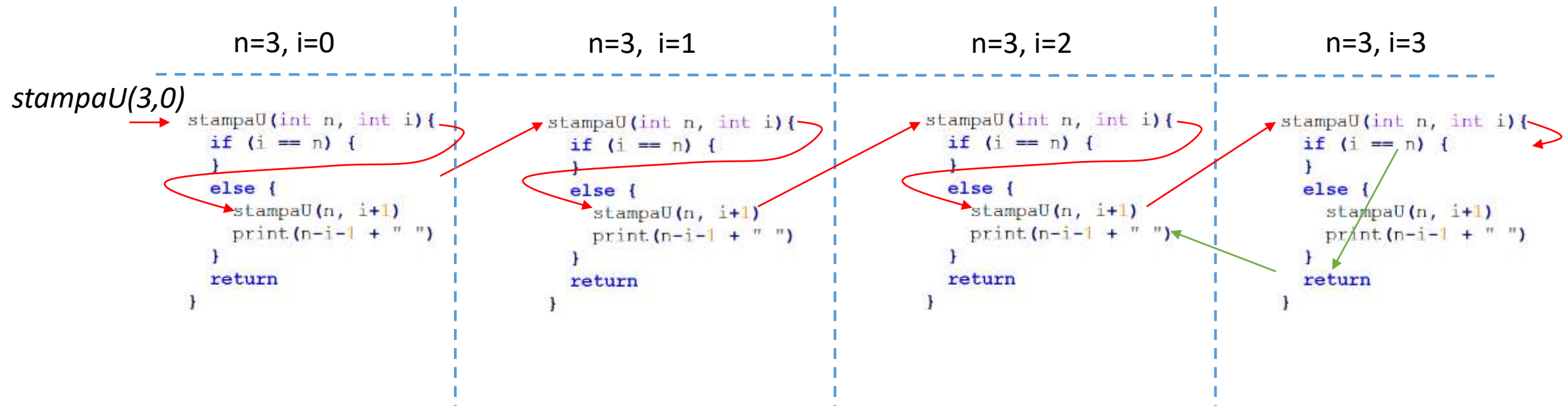
- Esempio per $n=3$



Ricorsione controvariante crescente $[0, \dots, n)$

```
>java StampaSegmConFinaleTest
```

- Esempio per $n=3$

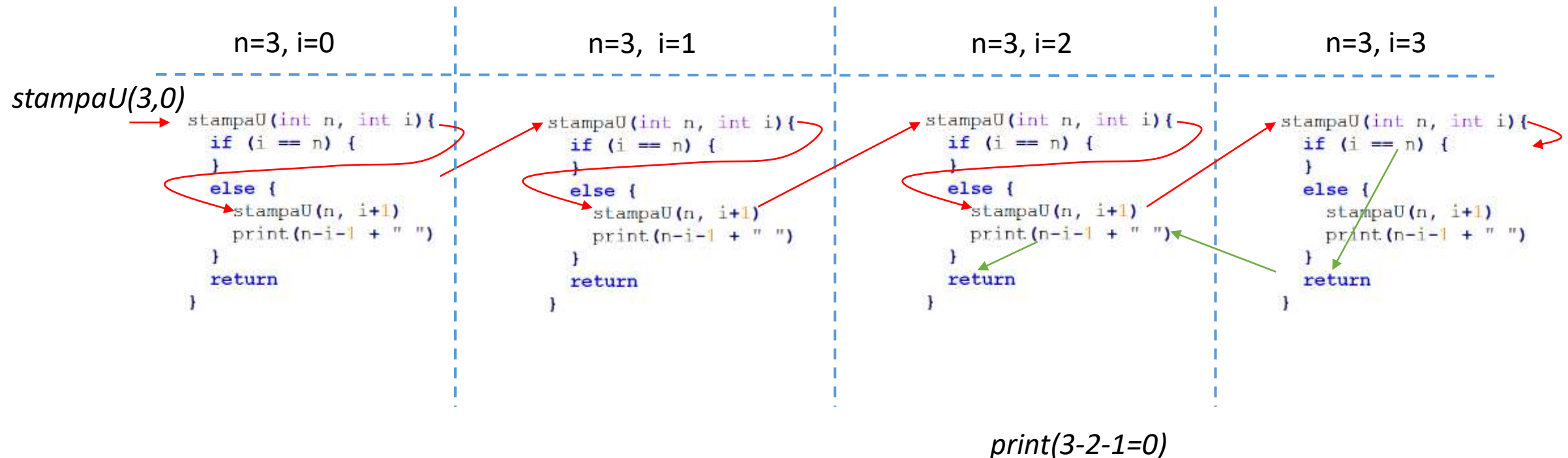


Ritorno a `stampaU(3,2)`

Ricorsione controvariante crescente $[0, \dots, n)$

- Esempio per $n=3$

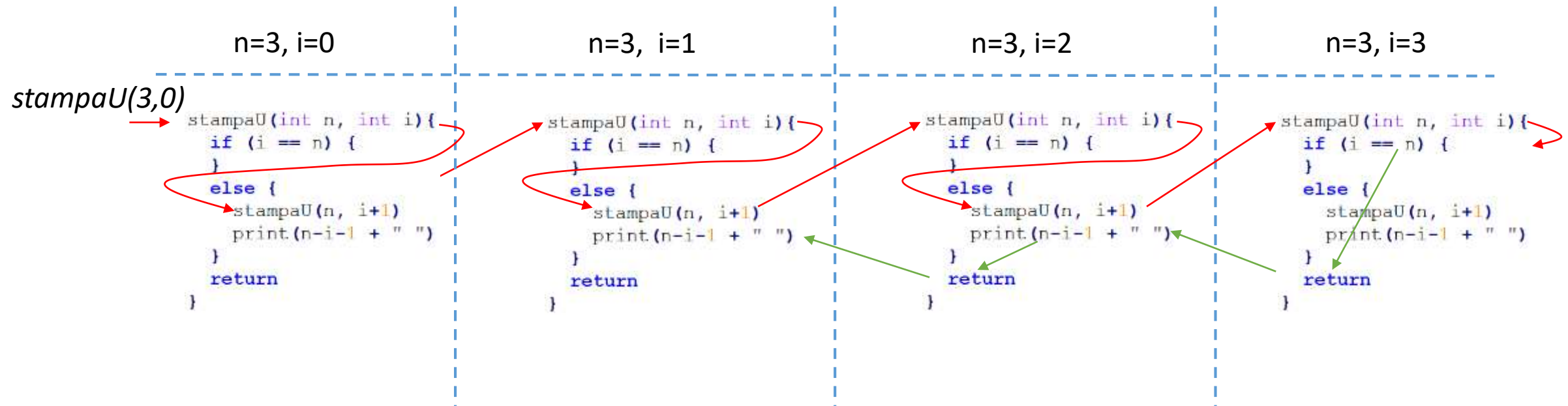
```
>java StampaSegmConFinaleTest  
0
```



Ricorsione controvariante crescente $[0, \dots, n)$

- Esempio per $n=3$

```
>java StampaSegmConFinaleTest  
0
```

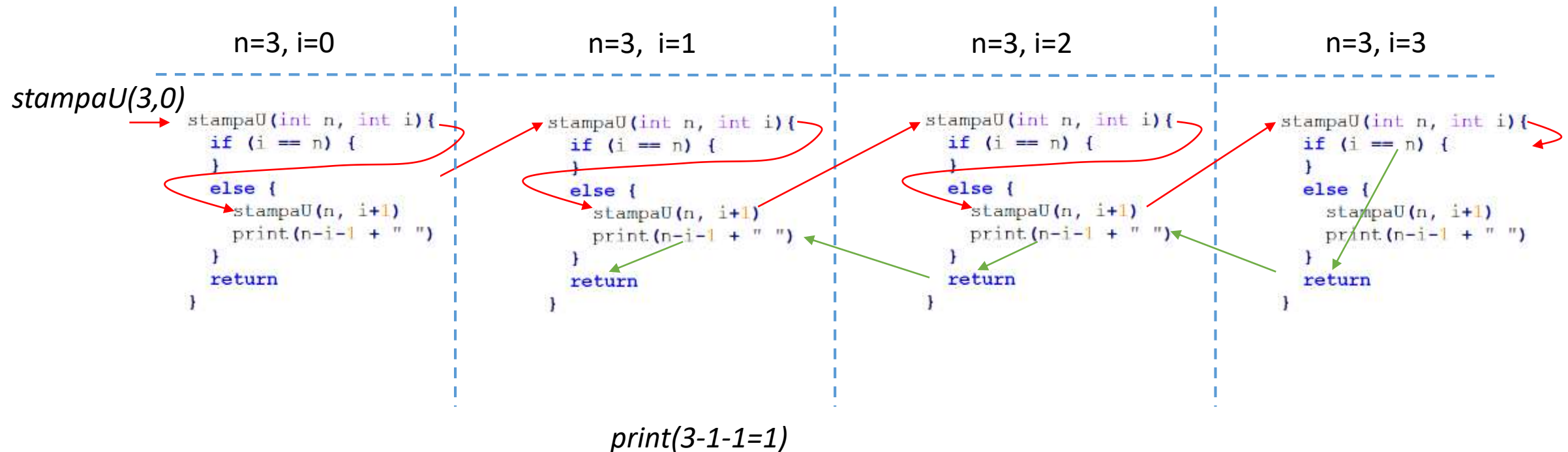


Ritorno a `stampaU(3,1)`

Ricorsione controvariante crescente $[0, \dots, n)$

- Esempio per $n=3$

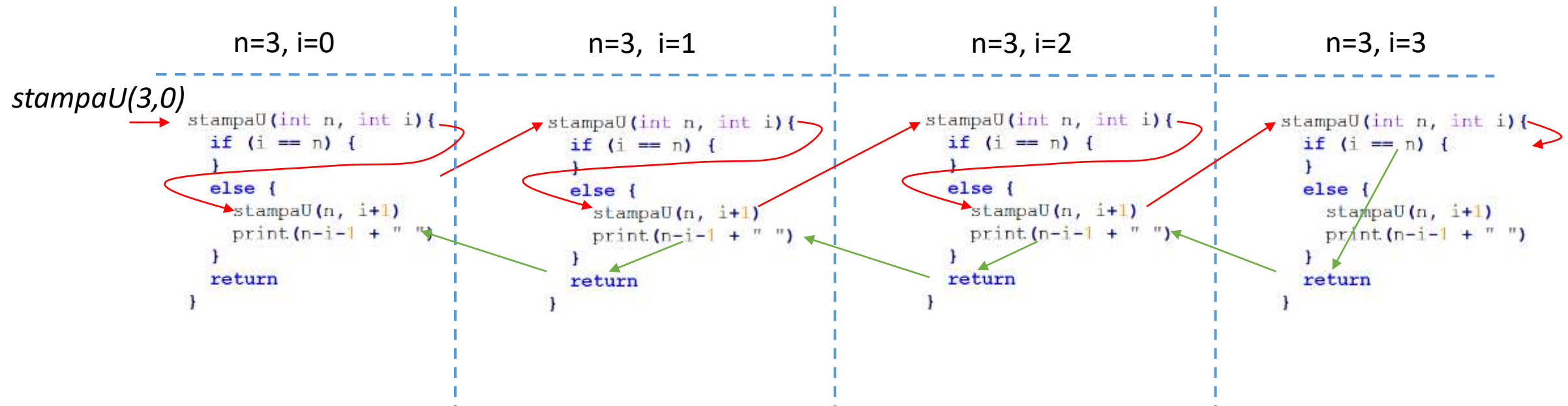
```
>java StampaSegmConFinaleTest  
0 1
```



Ricorsione controvariante crescente $[0, \dots, n)$

- Esempio per $n=3$

```
>java StampaSegmConFinaleTest  
0 1
```

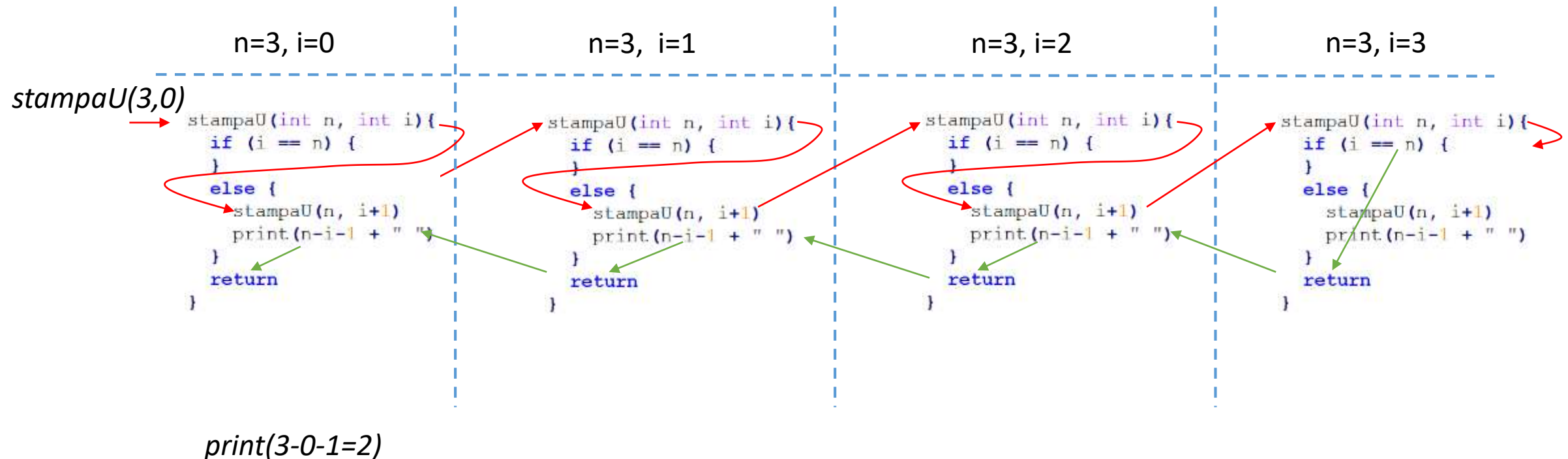


Ritorno a `stampaU(3,0)`

Ricorsione controvariante crescente $[0, \dots, n)$

- Esempio per $n=3$

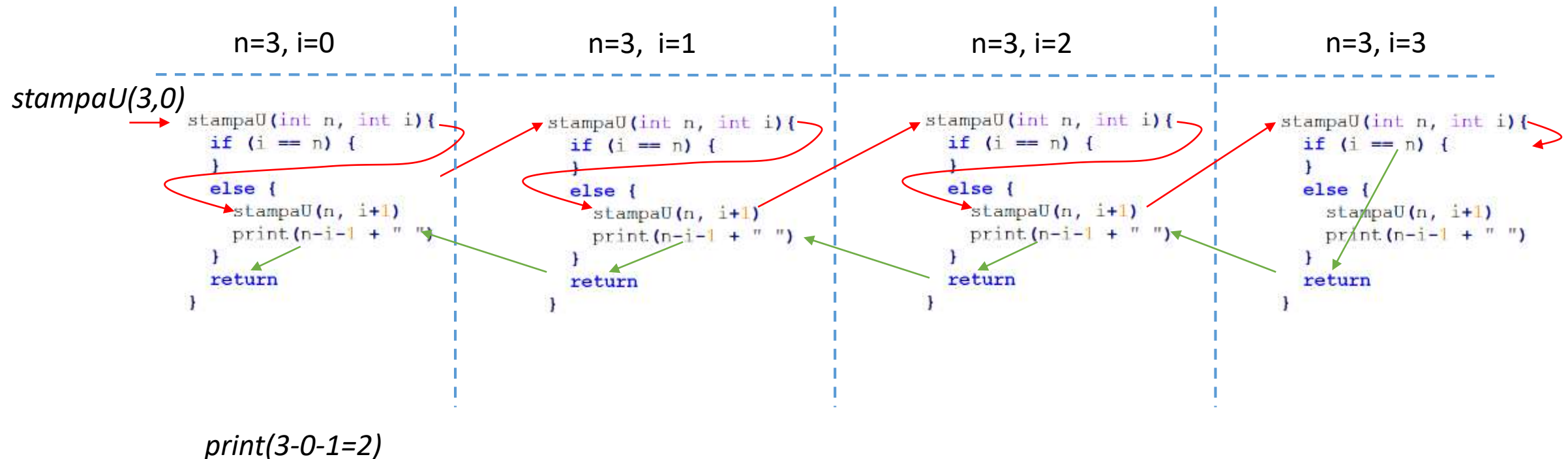
```
>java StampaSegmConFinaleTest  
0 1 2
```



Ricorsione controvariante crescente $[0, \dots, n)$

- Esempio per $n=3$

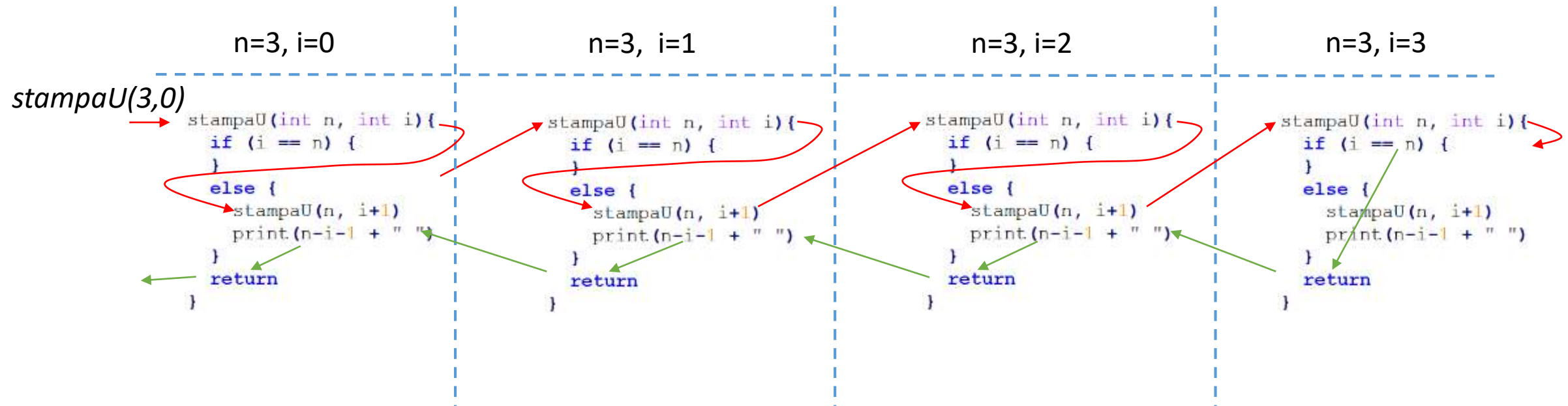
```
>java StampaSegmConFinaleTest  
0 1 2
```



Ricorsione controvariante crescente $[0, \dots, n)$

- Esempio per $n=3$

```
>java StampaSegmConFinaleTest  
0 1 2
```



return al wrapper

Ricorsione controvariante decrescente $[0, n)$

- Stampiamo i numeri naturali $[0, \dots, n)$ in ordine decrescente con un algoritmo controvariante per il caso $n=3$

```
public class StampaSegmConFinaleTest {  
  
    public static void main(String[] args){  
        System.out.println("--- Test stampaD");  
  
        StampaSegmConFinale.stampaD(3);System.out.println();  
    }  
}
```

↑
stampaU() chiamato
prima volta per $n = 3$

Ricorsione controvariante crescente $[0, \dots, n)$

```
/* Wrapper del metodo controvariante */
public static void stampaD(int n){
    stampaD(n, 0);
}

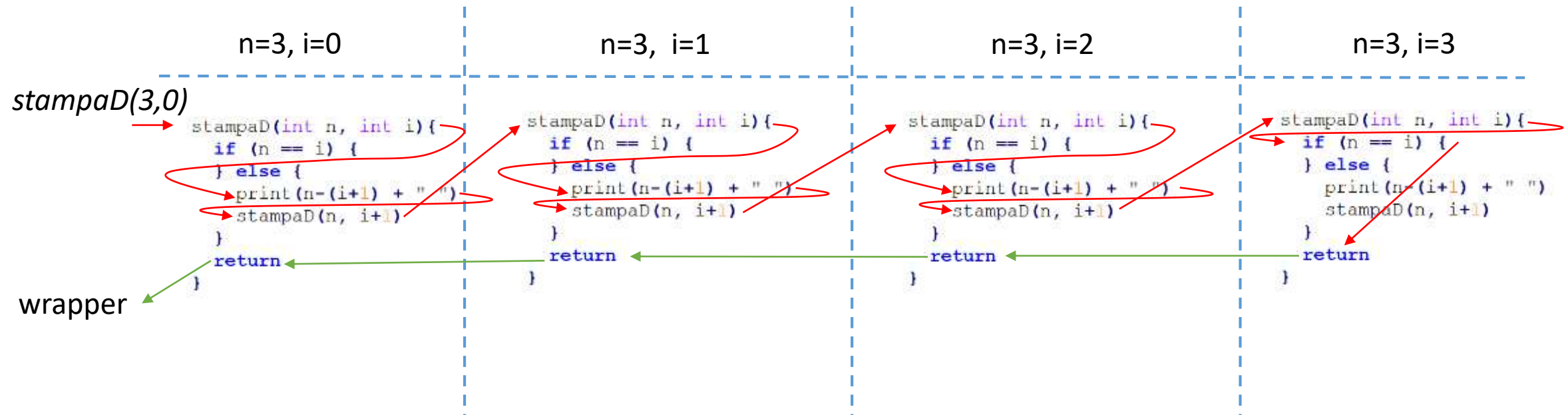
/* Metodo controvariante che stampa (n, 0] */
public static void stampaD(int n, int i){
    if (n == i) {
        /* Stampa (n-n, 0], cioè (0, 0], che è vuoto */
    } else {
        System.out.print(n - (i+1) + " "); /* Stampa n-i-1 */
        stampaD(n, i+1); /* Stampa (n-i-1, 0] */
    }
    return;
}
```

Overloading di *stampaD()*

Ricorsione controvariante decrescente (n, 0]

- Esempio per $n=3$

```
>java StampaSegmConFinaleTest  
2 1 0
```



Esercizi sulla ricorsione

Esercizio: Ricorsione fattoriale

- Si implementi la funzione fattoriale

$$n! = \prod_{k=1}^n k = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n$$

con un algoritmo ricorsivo covariante ricordando che

$$n! = n \times (n-1)! .$$

- Si faccia riferimento ai due casi seguenti

$$x! = \begin{cases} 1 & \text{se } x = 0 \\ x \times (x-1)! & \text{se } x > 0 \end{cases} .$$


Esercizio: Ricorsione fattoriale

```
public static int fattorialeRic(int n) {  
    int ret = 0;  
    /* CAso base */  
    if (n == 0) {  
        ret = 1;  
    }  
    /* Caso generale */  
    else {  
        ret = n * fattorialeRic(n - 1);  
    }  
    return ret;  
}
```

Esercizio: MultipliRicorsivi

- Siano m ed n due numeri naturali, definire (e poi simulare con Java Visualizer) due metodi ricorsivi, co-variante e contro-variante, che stampino in ordine crescente e decrescente i primi n multipli non nulli di m . Ad esempio, se $m = 4$ e $n = 3$, allora verranno stampati i valori [12, 8, 4].

Ovvero, per $n > 0$



MultipliRicorsivi – covariante crescente

Escludo $n == 0$

```
public static void multipliU(int m, int n) {  
    if (n == 0) {  
        /* Non stampa multipli nulli di m */  
    } else {  
        multipliU(m, n - 1); /* Stampa m 2m .. (n-1)m */  
        System.out.println(n*m + " "); /* Stampa nm */  
        /* Stampa m 2m .. (n-1)m nm */  
    }  
    return;  
}
```

MultipliRicorsivi – covariante decrescente

Escludo n == 0

```
public static void multipliD(int m, int n) {  
    if (n == 0) {  
        /* Non stampa multipli nulli di m */  
    } else {  
        System.out.println(n*m + " "); /* Stampa nm */  
        multipliU(m, n - 1); /* Stampa (n-1)m .. 2m m */  
        /* Stampa nm (n-1)m .. 2m m */  
    }  
    return;  
}
```

```
public static void main(String[] args) {  
    multipliU(4, 3);  
    System.out.println();  
    multipliD(4, 3);  
}
```

MultipliRicorsivi – controvariante crescente

```
public static void multipliU(int m, int n, int i) {  
    if (n == i) {  
        /* Non stampa multipli nulli di m */  
    } else {  
        multipliU(m, n, i + 1); /* Stampa m 2m .. (n-(i+1))m  
                                Cioè stampa m 2m .. (n-i-1)m */  
        System.out.println((n-i)*m + " "); /* Stampa (n-i)m */  
        /* Stampa m 2m .. (n-i-1)m (n-i)m */  
    }  
    return;  
}  
  
public static void multipliU(int m, int n) {  
    multipliU(m, n, 0);  
}
```

MultipliRicorsivi – controvariante decrescente

```
public static void multipliD(int m, int n, int i) {  
    if (n == i) {  
        /* Non stampa multipli nulli di m */  
    } else {  
        System.out.println((n-i)*m + " "); /* Stampa (n-i)m */  
        multipliD(m, n, i + 1); /* Stampa (n-(i+1))m .. 2m m */  
        /* Stampa (n-i)m (n-(i+1))m .. 2m m */  
    }  
    return;  
}  
  
public static void multipliD(int m, int n) {  
    multipliD(m, n, 0);  
}
```

Esercizio: AritmeticaRic – per casa

- Realizzare una classe *AritmeticaRic.java*, e relativa *AritmeticaRicTest.java*, con metodi ricorsivi co-varianti e contro-varianti che calcolano le operazioni:
 - somma
 - differenza
 - moltiplicazione
 - potenza
 - quoziente
 - resto

Esercizi ricorsione su array

ESERCIZIO ArrayRic – stampaArrayDecCoX()

- Scrivere un metodo ricorsivo *stmpaArrayDec()* che, dato un array di interi a , ne stampi il contenuto in ordine di indice decrescente (ovvero, dall'ultimo al primo elemento dell'array)
- Se ne implementi una versione covariante ed una controvariante
- Si implementino i relativi metodi wrapper

ESERCIZIO ArrayRic – stampaArrayDecCov()

```
/* Stampa il contenuto di un array in ordine decrescente */
public static void stampaArrayDecCov(int[] a, int n) {
    // Caso base: stampo solo il carattere "a capo"
    if (n == -1){
        System.out.println("");
        return;
    }
    else {
        System.out.print(a[n] + " ");
        stampaArrayDecCov(a, n - 1);
        return;
    }
}

/* Metodo wrapper */
public static void stampaArrayDecCov(int [] a) {
    if (a != null) {
        stampaArrayDecCov(a, a.length - 1);
    }
}
```


ESERCIZIO ArrayRic – stampaArrayDecCon()

```
/* Stampa il contenuto di un array in ordine decrescente */
public static void stampaArrayDecCon(int[] a, int n, int i) {
    // Caso base: stampo solo il carattere "a capo"
    if (i == n){
        return;
    }
    else {
        stampaArrayDecCon(a, n, i + 1);
        System.out.print(a[i] + " ");
        return;
    }
}

/* Metodo wrapper */
public static void stampaArrayDecCon(int [] a) {
    if (a != null) {
        stampaArrayDecCon(a, a.length, 0);
    }
}
```

ESERCIZIO ArrayRic – stringifyRic()

- Scrivere un metodo *stringfyRic()* che, dato un array di interi a , ne restituisca la rappresentazione in String, separandone gli elementi con spazi. Per esempio, dato l'array

```
int[] x = {10, 20, 30, 40};
```

il metodo dovrà restituire (si noti lo spazio al termine della stringa)

«10 20 30 40 »

- Si utilizzi l'operatore di concatenazione fra stringhe «+»

ESERCIZIO ArrayRic – stringifyRic()

```
/* Concatena il contenuto di un array in una stringa */
public static String stringifyRic(int[] a, int n, String s) {
    // Caso base: bordo estremo sinistro dell'array a, nulla da concatenare
    if (n == 0){
    }
    else {
        s = stringifyRic(a, n-1, s);
        s = s + a[n-1] + " ";
    }
    return s;
}

/* Metodo wrapper */
public static String stringifyRic(int [] a) {
    String ret = "";
    if (a != null) {
        ret = stringifyRic(a, a.length, ret);
    }
    return ret;
}
```

ESERCIZIO ArrayRic – esisteRic()

- Scrivere un metodo *esisteRic()* che, dati un array di interi *a* ed un intero *m*, ritorni *true* se *a* contiene *m*, *false* altrimenti. Si implementi il metodo con un algoritmo ricorsivo gestendo opportunamente il caso in cui *a* sia *null* in un metodo wrapper e si verifichi il funzionamento del metodo per l'array *a* fornito nel test case e per i valori di *m* {0, 10, 40}.

ESERCIZIO ArrayRic – esisteRic()

```
/* Ritorna true se m è presente in a, false altrimenti */
public static boolean esisteRic(int[] a, int m, int n) {
    //Caso base, volutamente lasciato vuoto
    if (n == 0) {
        return false;
    } else {
        // Implementa return (a[n-1] == m) || esiste2(a, n-1);
        // notare l'accesso ad a[n-1] anzichè a[n]
        if (a[a.length-n] == m)
            return true;
        else
            return esisteRic(a, m, n-1);
    }
}

/* Metodo wrapper */
public static boolean esisteRic(int[] a, int m) {
    boolean ret = false;
    if (a != null) {
        ret = esisteRic(a, m, a.length);
    }
    return ret;
}
```

ESERCIZIO ArrayRic – esisteRicPos()

- Scrivere un metodo *esisteRicPos()* che, dati un array di interi a ed un intero m , ritorni la prima posizione di m in a , la lunghezza di a altrimenti. Si implementi il metodo con un algoritmo ricorsivo gestendo opportunamente il caso in cui a sia *null* in un metodo wrapper e si verifichi il funzionamento del metodo per l'array a fornito nel test case e per i valori di m $\{0, 10, 40\}$.

ESERCIZIO ArrayRic – esisteRicPos()

```
/* Ritorna la posizione di m in a, a.length altrimenti */
public static int esisteRicPos(int[] a, int m, int n) {
    //Caso base, volutamente lasciato vuoto
    if (n == 0) {
        return a.length;
    } else {
        // Implementa return (a[n-1] == m) || esiste2(a, n-1);
        // notare l'accesso ad a[n-1] anzichè a[n]
        if (a[n-1] == m)
            return n-1;
        else
            return esisteRicPos(a, m, n-1);
    }
}

/* Metodo wrapper */
public static int esisteRicPos(int[] a, int m) {
    int ret = -1;
    if (a != null) {
        ret = esisteRicPos(a, m, a.length);
    }
    return ret;
}
```

ESERCIZIO ArrayRic – tuttiPari()

- Scrivere un metodo *tuttiPari()* che, dato un array di interi *a*, ritorni *true* se tutti gli elementi di *a* sono pari, *false* altrimenti. Si implementi il metodo con un algoritmo ricorsivo gestendo opportunamente il caso in cui *a* sia *null* in un metodo wrapper e si verifichi il funzionamento del metodo per l'array *x* fornito nel test case.

ESERCIZIO ArrayRic – tuttiPari()

```
/* Ritorna true se tutti gli elementi dell'array sono pari, false altrimenti */
public static boolean tuttiPari(int[] a, int n) {
    if (n < 0) {
        // Caso base: ricorsione oltre il primo elemento di a
        return true;
    }
    else {
        return (a[n] % 2 == 0) && tuttiPari(a, n - 1);
    }
}

/* Metodo wrapper */
public static boolean tuttiPari(int[] a) {
    boolean ret = false;
    if (a != null) {
        ret = tuttiPari(a, a.length - 1);
    }
    return ret;
}
```

ESERCIZIO ArrayRic – esisteMultiplo()

- Scrivere un metodo *esisteMultiplo(a, n)* che, dato un array di interi *a* ed un intero *m*, ritorni *true* se *a* contiene un elemento multiplo di *m*, false altrimenti. Si implementi il metodo con un algoritmo ricorsivo controvariante gestendo opportunamente il caso in cui *a* sia *null* in un metodo wrapper e si verifichi il funzionamento del metodo per l'array a fornito nel test case.

ESERCIZIO ArrayRic – esisteMultiplo()

```
/* Ritorna true se a contiene un elemento multiplo di m. */
public static boolean esisteMultiplo(int[] a, int m, int n) {
    // Caso base: ricorsione oltre l'ultimo elemento di a
    if (n >= a.length) {
        return false;
    }
    else {
        return (a[n] % m == 0) || esisteMultiplo(a, m, n + 1);
    }
}

/* Metodo wrapper */
public static boolean esisteMultiplo(int[] a, int m) {
    boolean ret = false;

    if (a != null) {
        ret = esisteMultiplo(a, m, 0);
    }

    return ret;
}
```

ESERCIZIO ArrayRic – filtraMaggioriDi()

- Scrivere un metodo *filtraMaggioriDi(a, m)* che, dato un array di interi *a* ed un intero *m*, ritorni un **nuovo** array contenente solo gli elementi di *a* che sono strettamente maggiori di *m* in ordine inverso per semplicità. Si implementi il metodo con un algoritmo ricorsivo in modo che, ad ogni passo della ricorsione, si mantenga un conteggio degli elementi che hanno superato il test. Nel caso base si avrà a disposizione il numero di elementi filtrati e si allocherà l'array da ritornare. Si verifichi il funzionamento del metodo per l'array a fornito nel test case.

ESERCIZIO ArrayRic – filtraMaggioriDi()

```
/* Ritorna un nuovo array contenente solo gli elementi di a che sono
 * strettamente maggiori di m in ordine inverso.
 */
public static int[] filtraMaggioriDi(int[] a, int m, int n, int counter) {
    // caso base: alloca l'array per @counter elementi da memorizzare
    if (n < 0) {
        return new int[counter];
    }

    // caso generale
    if (a[n] > m) {
        // L'elemento a[n] viene copiato nell'array in uscita in posizione @counter.
        // Aumenta di 1 il conteggio, prosegui la ricorsione e poi
        // assegna a[n] nell'array da restituire
        int[] outArr = filtraMaggioriDi(a, m, n - 1, counter + 1);
        outArr[counter] = a[n];
        return outArr;
    }
    else // L'elemento a[n] non viene copiato nell'output
        return filtraMaggioriDi(a, m, n - 1, counter);
}
```