Algoritmi ricorsivi
Paradigma *divide et impera*
Recursive binary search
Merge Sort

Liceo G.B. Brocchi
Classi seconde Scientifico – opzione scienze applicate
Bassano del Grappa, Aprile 2023
Prof. Giovanni Mazzocchin

L’approccio *divide, conquer and combine*

- Molti problemi sono risolvibili «naturalmente» in modo ricorsivo. Con *naturalmente* intendo dire che per questi problemi è più immediato trovare una soluzione ricorsiva rispetto ad una iterativa
- Questi algoritmi ricorsivi, per risolvere un problema, chiamano sé stessi ricorsivamente su un certo numero di sottoproblemi almeno una volta. Questi procedimenti sono matematicamente validi in quando i sottoproblemi *assomigliano* al problema di partenza

- L’approccio che prevede la suddivisione di un problema in sottoproblemi (e la loro risoluzione ricorsiva) viene detto ***divide and conquer*** (**in latino *divide et impera***)

Divide, conquer and combine

Il paradigma *divide and conquer* è composto da tre passaggi

1. dividi il problema in un certo numero di sottoproblemi

2. conquista i sottoproblemi, ossia risolvili ricorsivamente
Se i problemi sono di dimensione minima (casi base), non suddividerli più e risolvili direttamente

3. combina le soluzioni dei sottoproblemi e genera la soluzione del problema di partenza

La ricerca binaria ricorsiva

- Conosciamo già un algoritmo fondato sul paradigma *divide and conquer* (senza *combine*): la ricerca binaria
- Lo avevamo implementato iterativamente
- Vedremo che è più facile scriverlo ricorsivamente, in quanto la natura di questo algoritmo è intrinsecamente ricorsiva
- Scriviamolo prima in pseudocodice
- Essendo un algoritmo di ricerca, deve restituirci l'indice dell'elemento trovato, oppure -1 se l'elemento non è stato trovato
- La versione ricorsiva, in quanto esempio di *tail recursion*, sarà una specie di *duale* della versione iterativa: praticamente, la negazione della condizione di permanenza del ciclo costituirà il caso base della funzione ricorsiva!

La ricerca binaria ricorsiva: pseudocodice dell'implementazione iterativa

```
binary_search_I (A, low, high, key): returns bool
    bool found = false
    index = -1
    while (found == false AND low <= high):
        middle = (low + high) / 2
        if key == A[middle]:
            found = true
        else if key < A[middle]:
            high = middle - 1
        else
            low = middle + 1

    return found
```

Esempio di programmazione strutturata: nessun return o break all'interno del ciclo

Destrutturiamolo un po' per arrivare alla versione ricorsiva!

Praticamente, proviamo a scriverlo male, ma veramente male

La ricerca binaria ricorsiva: pseudocodice dell'implementazione iterativa

```
binary_search_I(A, low, high, key): returns bool
    while (low <= high):
        middle = (low + high) / 2
        if key == A[middle]:
            return true
        if low > high:
            return false
        if key < A[middle]:
            high = middle - 1
        else
            low = middle + 1
```

Vi ricordo che stiamo facendo un esempio per arrivare alla versione ricorsiva

È vietatissimo scrivere codice in questo modo, almeno quando si impara a programmare

La ricerca binaria ricorsiva: pseudocodice dell'implementazione iterativa

```
binary_search_R(A, low, high, key): returns bool
    if low > high:
        return false

    middle = (low + high) / 2

    if key == A[middle]:
        return true

    else if key < A[middle]:
        return binary_search(A, low, middle - 1, key)

    return binary_search(A, middle + 1, high, key)
```

Merge Sort (*John von Neumann, 1945*, *https://en.wikipedia.org/wiki/John_von_Neumann*)

- Vi sembrerà strano, ma possiamo ordinare una lista di elementi utilizzando un approccio ***divide, conquer and combine***
- L'algoritmo di ordinamento **Merge Sort**, può essere descritto così, informalmente:
  - **Divide**: dividi la lista di n elementi in 2 liste, ciascuna di n / 2 elementi
  - **Conquer**: ordina le 2 sottoliste ricorsivamente
  - **Combine**: fondi (*merge*) le 2 sottoliste ordinate

- Sembra un cane che si morde la coda, ma non lo è perché ad un certo punto si arriva a liste di dimensione 1, che non sono più suddivisibili e sono già ordinate

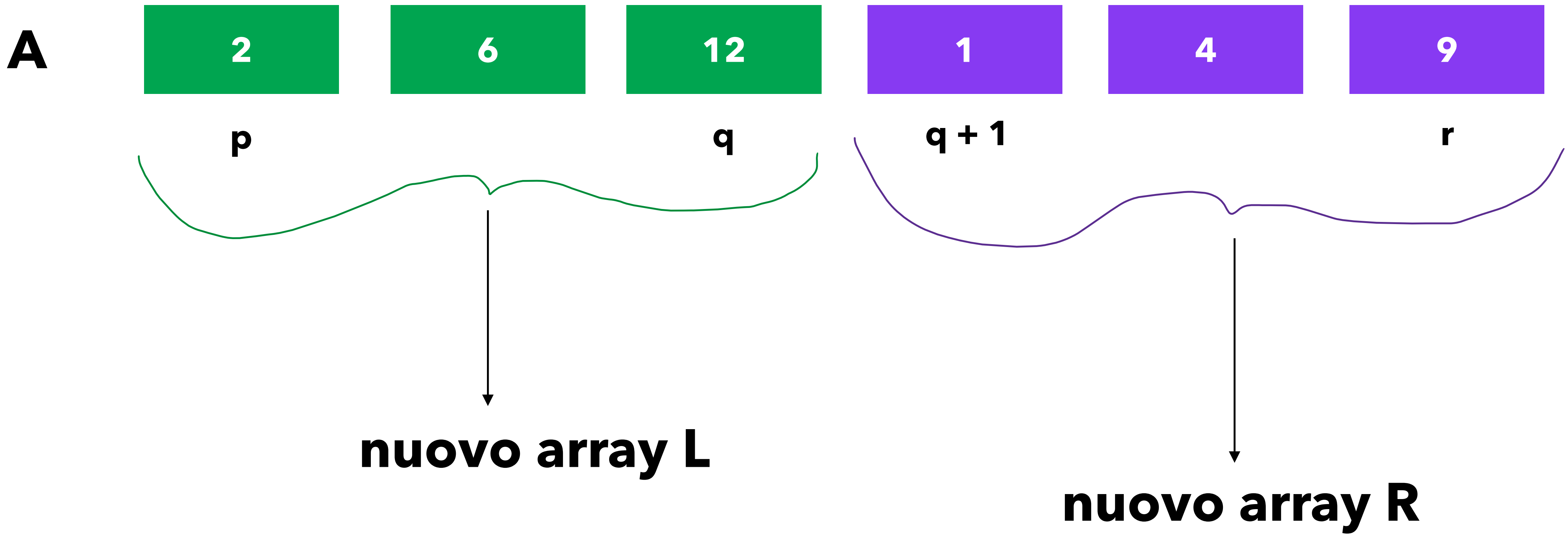
La procedura *merge*

- Prima di scrivere l'algoritmo merge sort, abbiamo bisogno di definire la procedura **`merge(A, p, q, r)`**, che, si comporta così:

```
dato un array A e tre con indici p <= q < r, per il quale:
     il sottoarray A[p..q] è ordinato
     il sottoarray A[q + 1, r] è ordinato

→ modifica A in modo da rendere A[p..r] ordinato
```

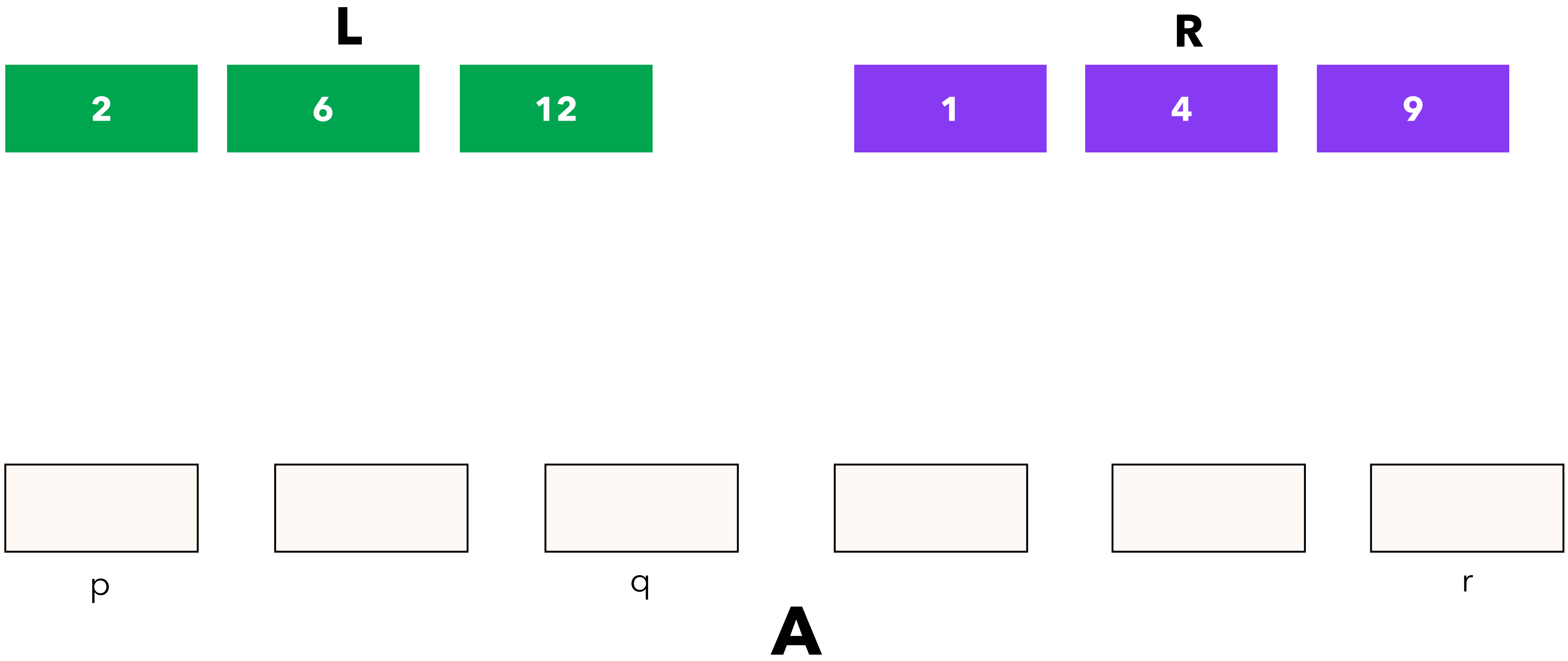
La procedura *merge* (esempio 1)



mergiamo L e R nell'array A

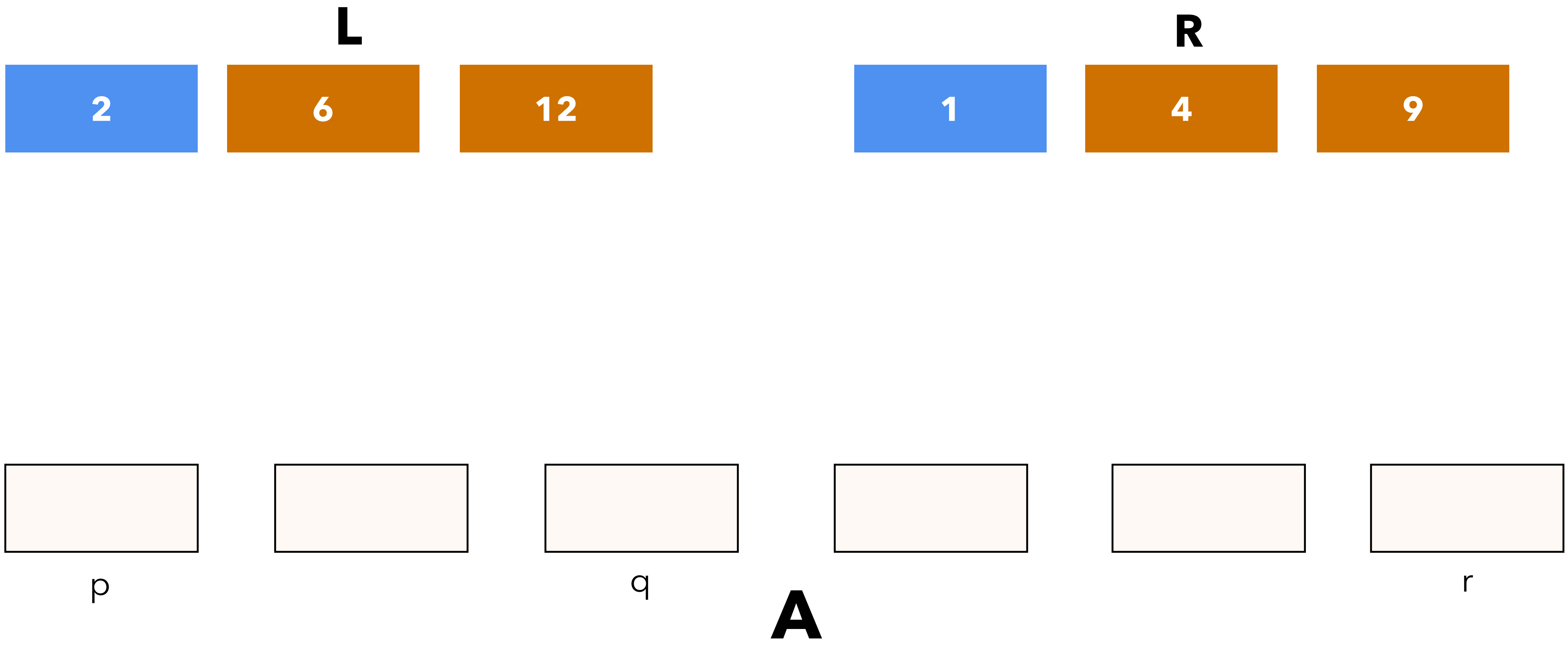
NB: gli elementi da fondere sono r – p + 1

La procedura *merge* (esempio 1)



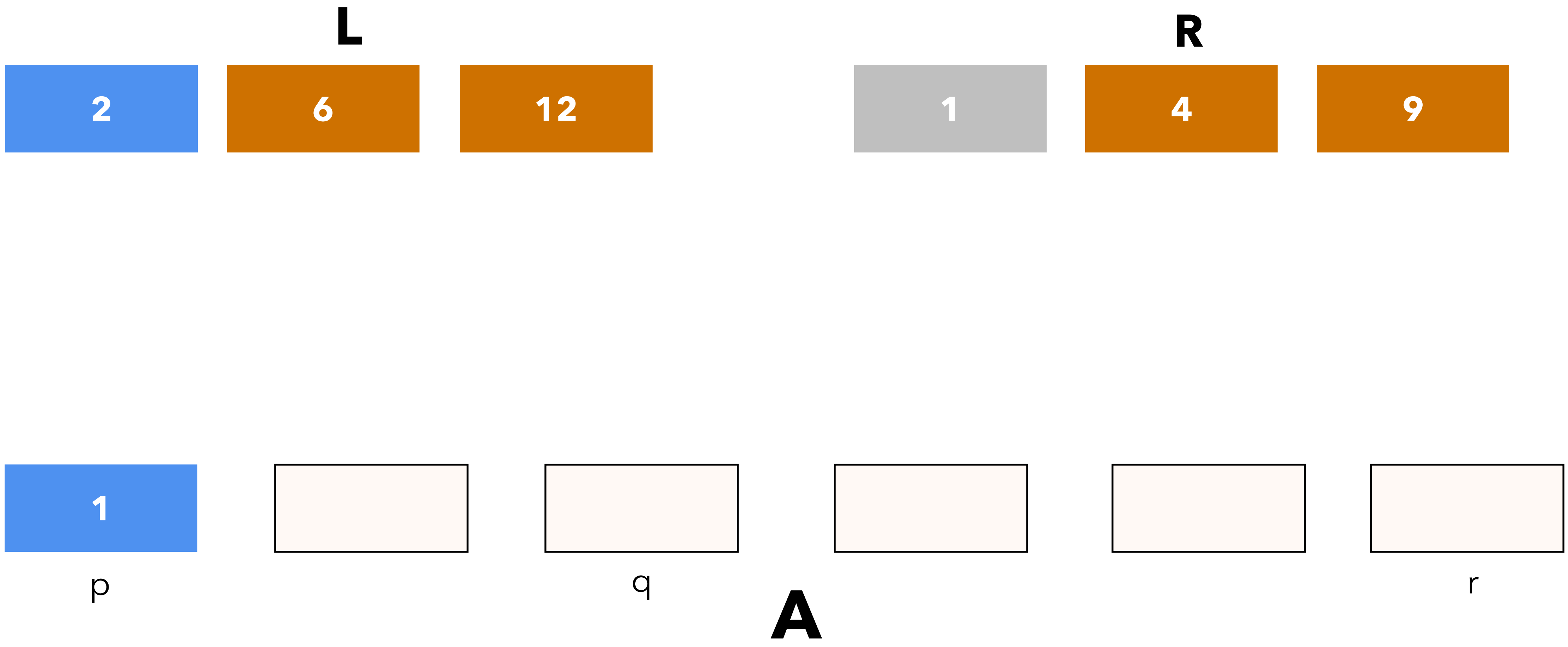
A* va riempito da p a r con gli array L e R *mergiati

La procedura *merge* (esempio 1)



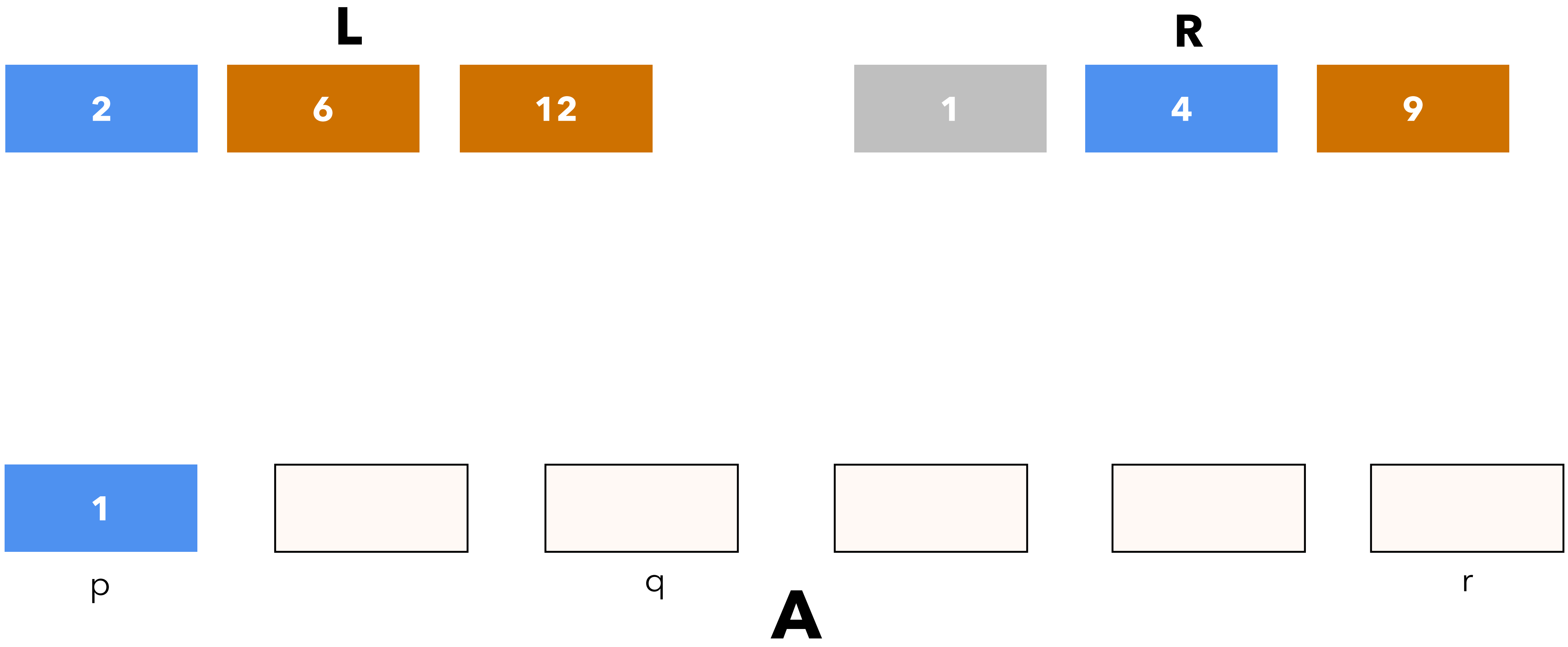
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A. In seguito non verrà più considerato, in quanto già «sistemato»

La procedura *merge* (esempio 1)



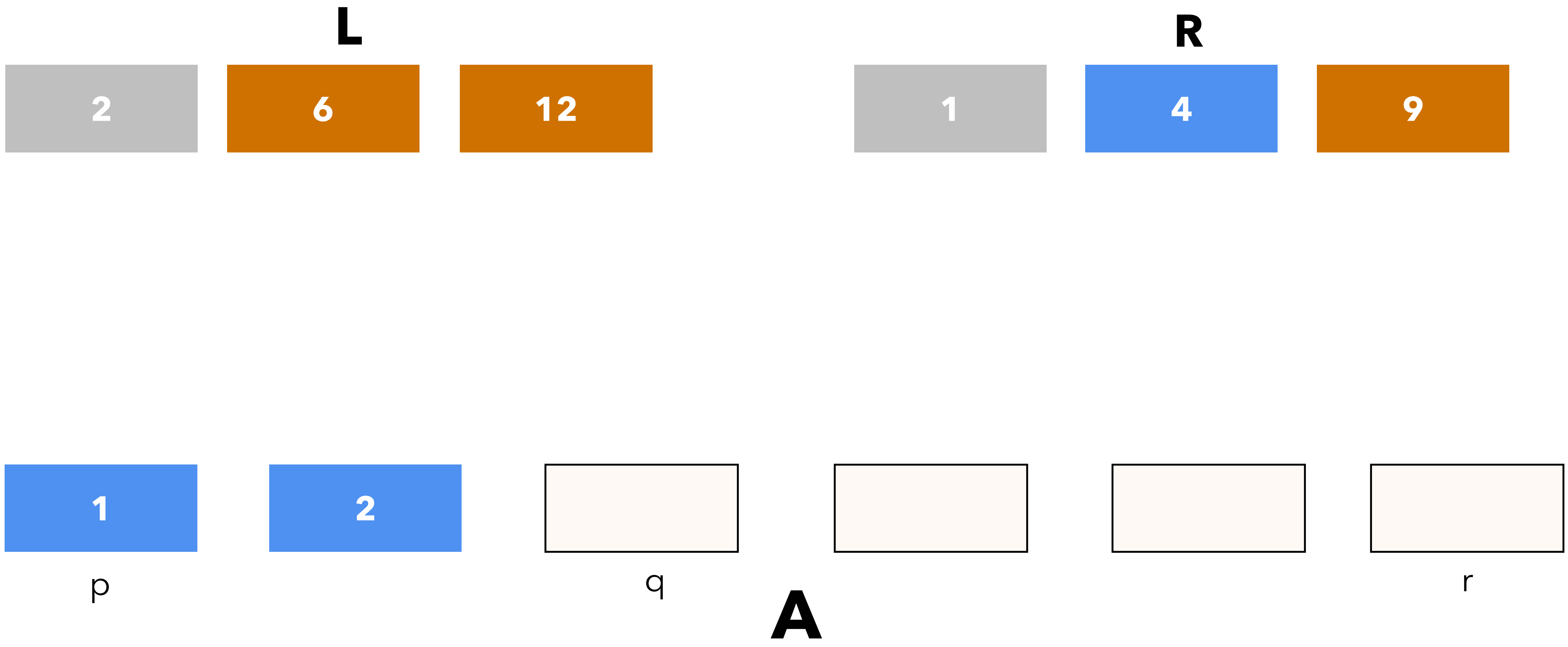
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A. In seguito non verrà più considerato, in quanto già «sistemato»

La procedura *merge* (esempio 1)



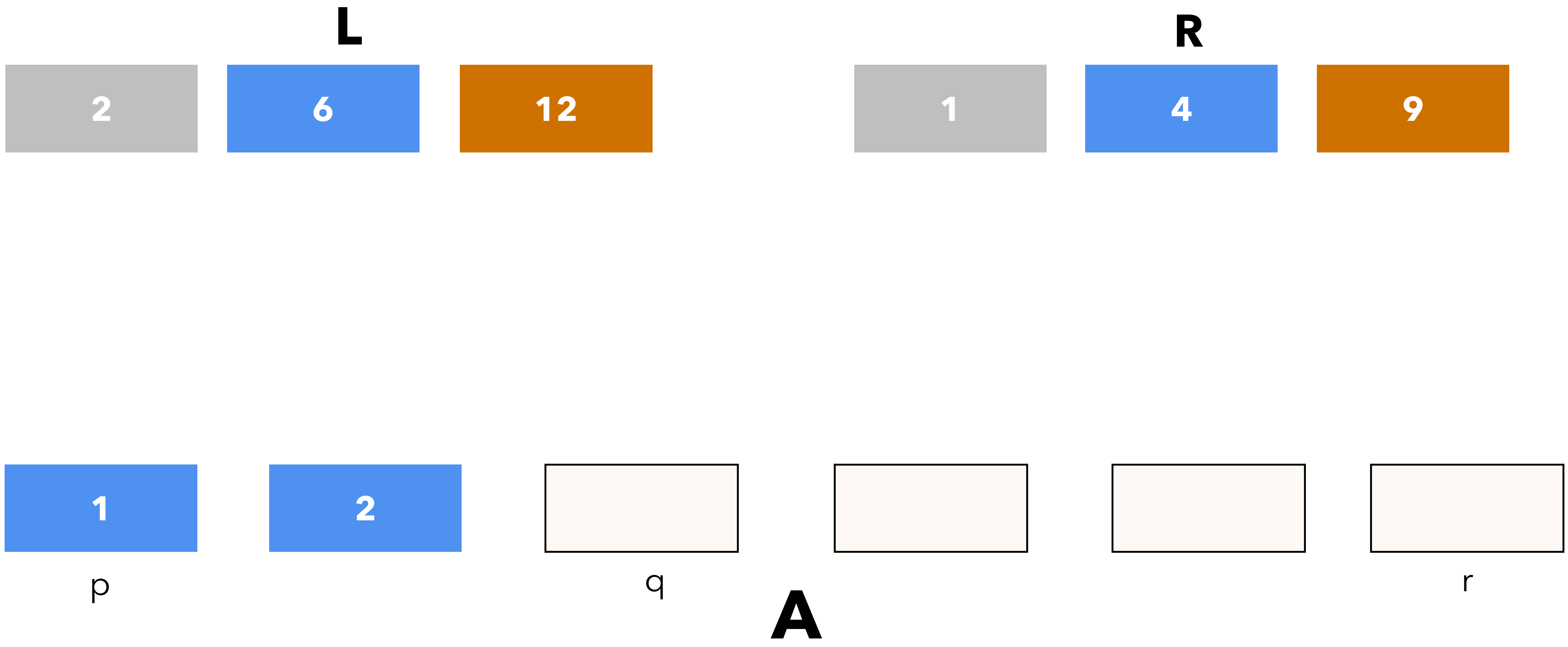
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

La procedura *merge* (esempio 1)



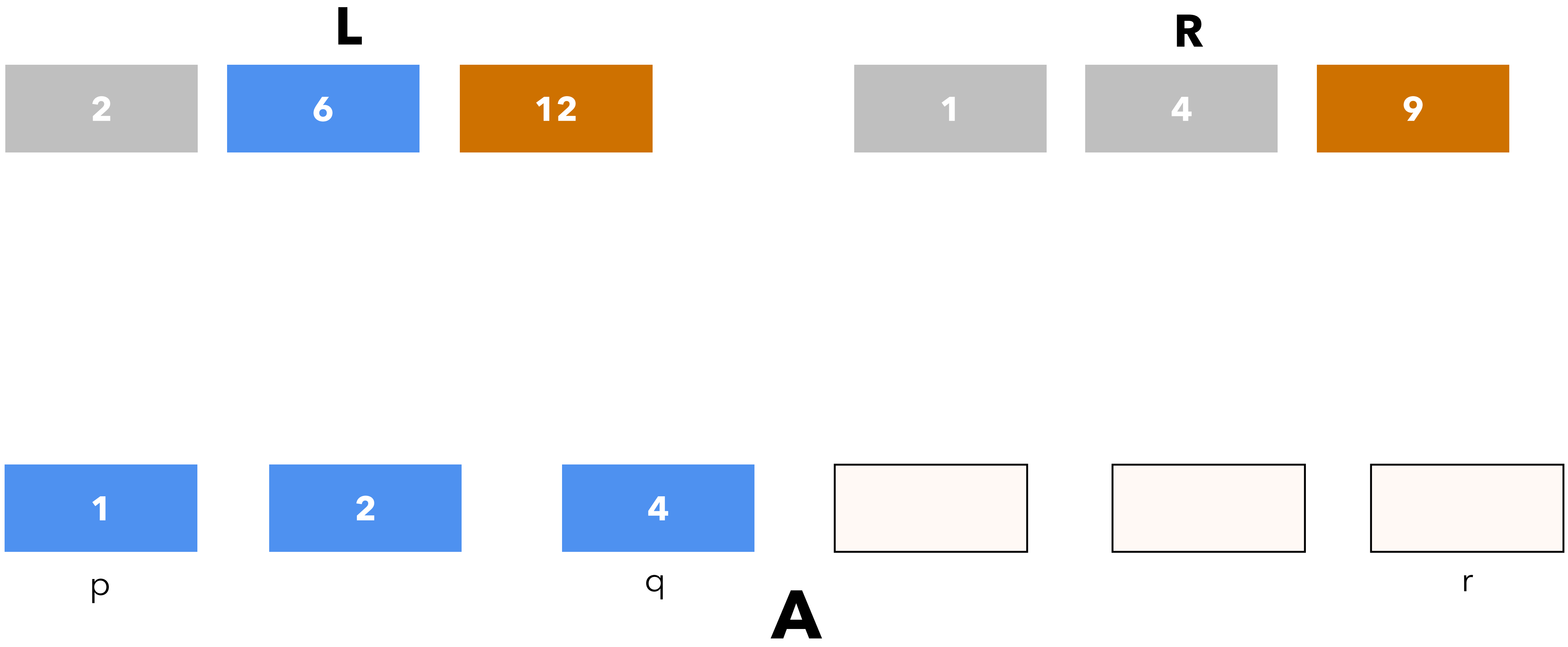
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

La procedura *merge* (esempio 1)



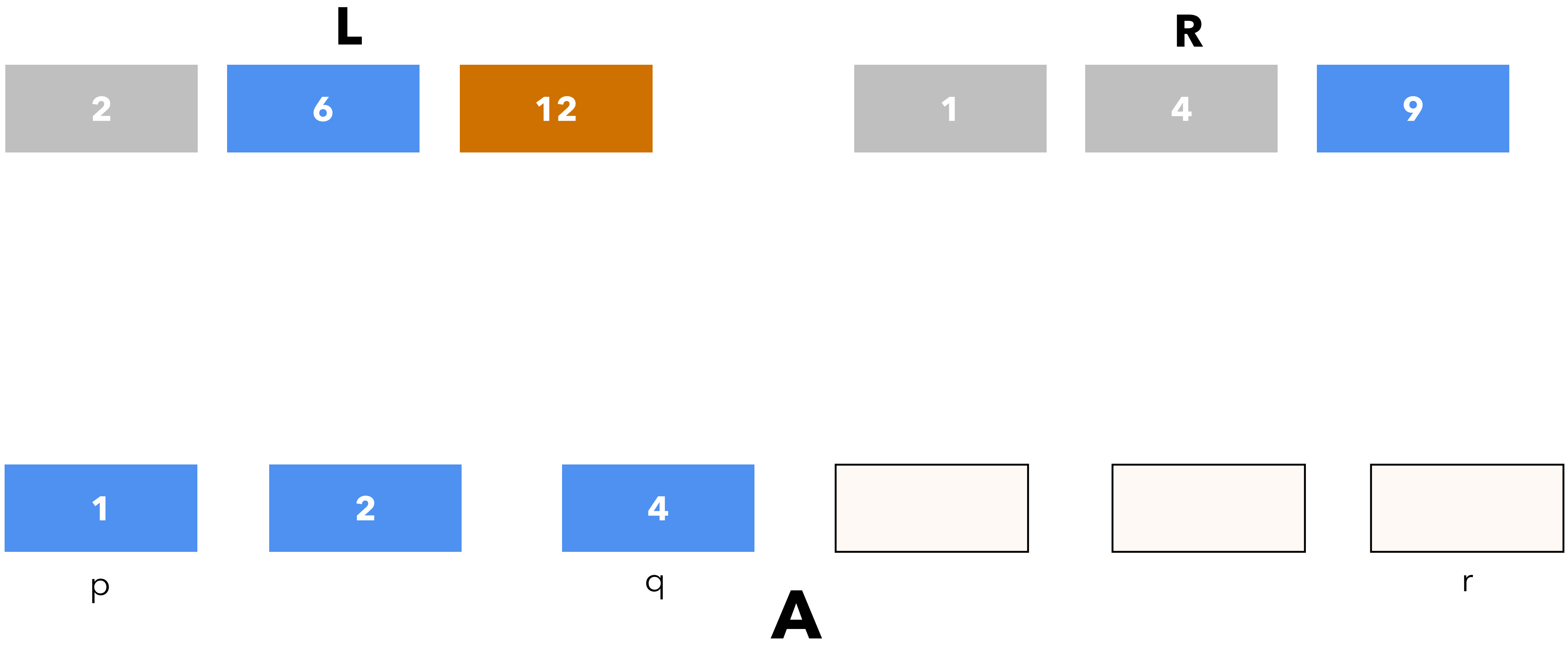
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

La procedura *merge* (esempio 1)



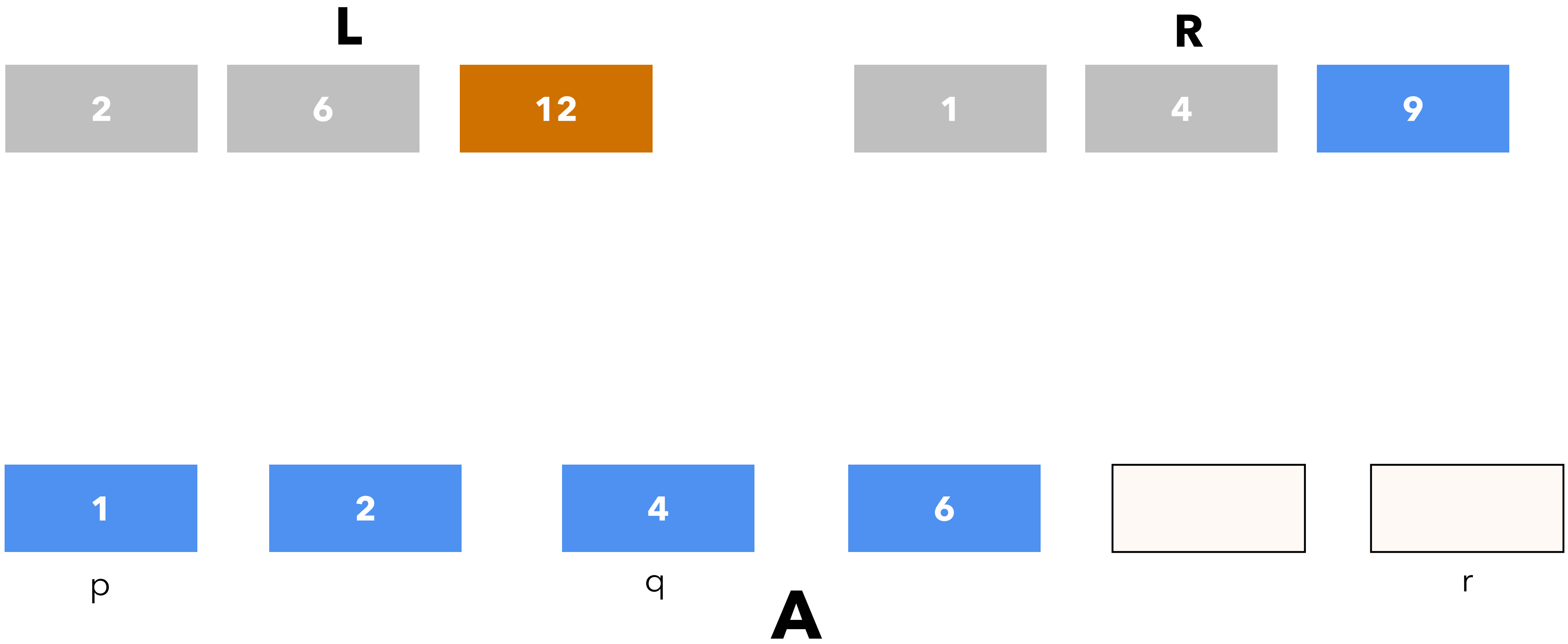
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

La procedura *merge* (esempio 1)



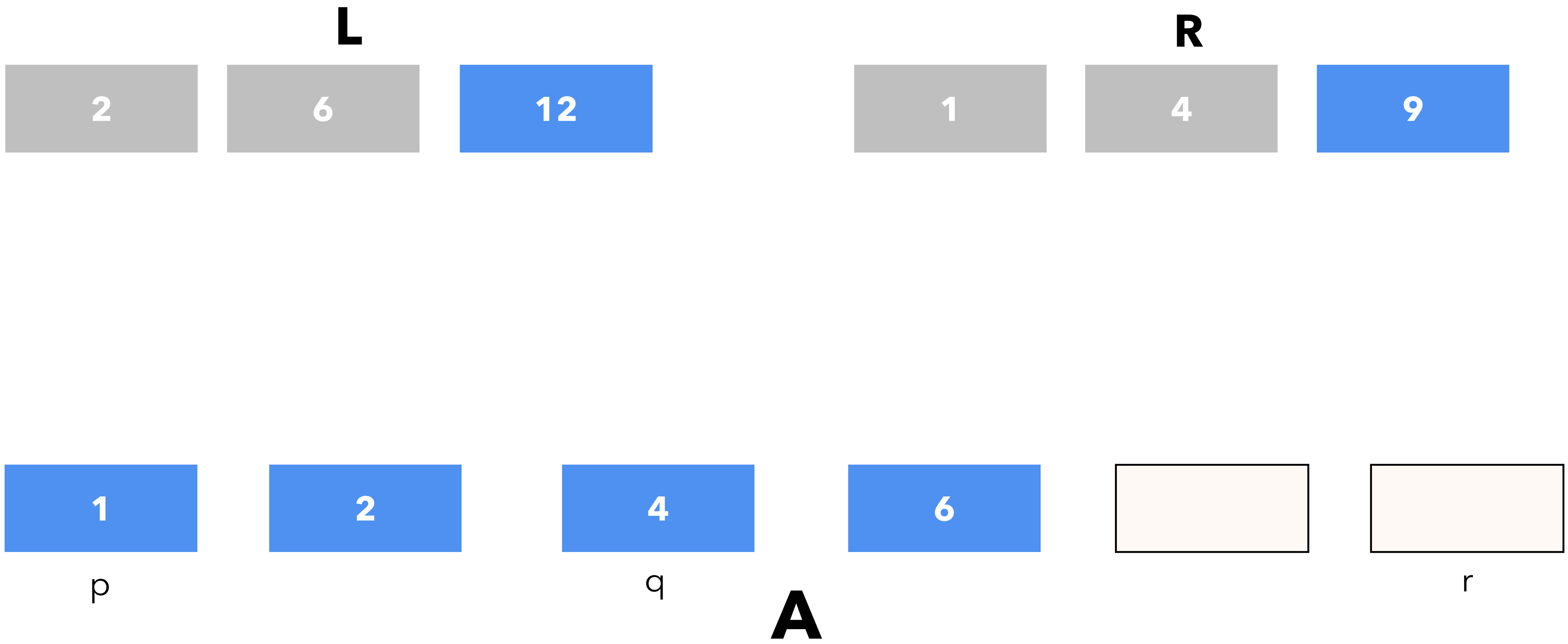
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

La procedura *merge* (esempio 1)



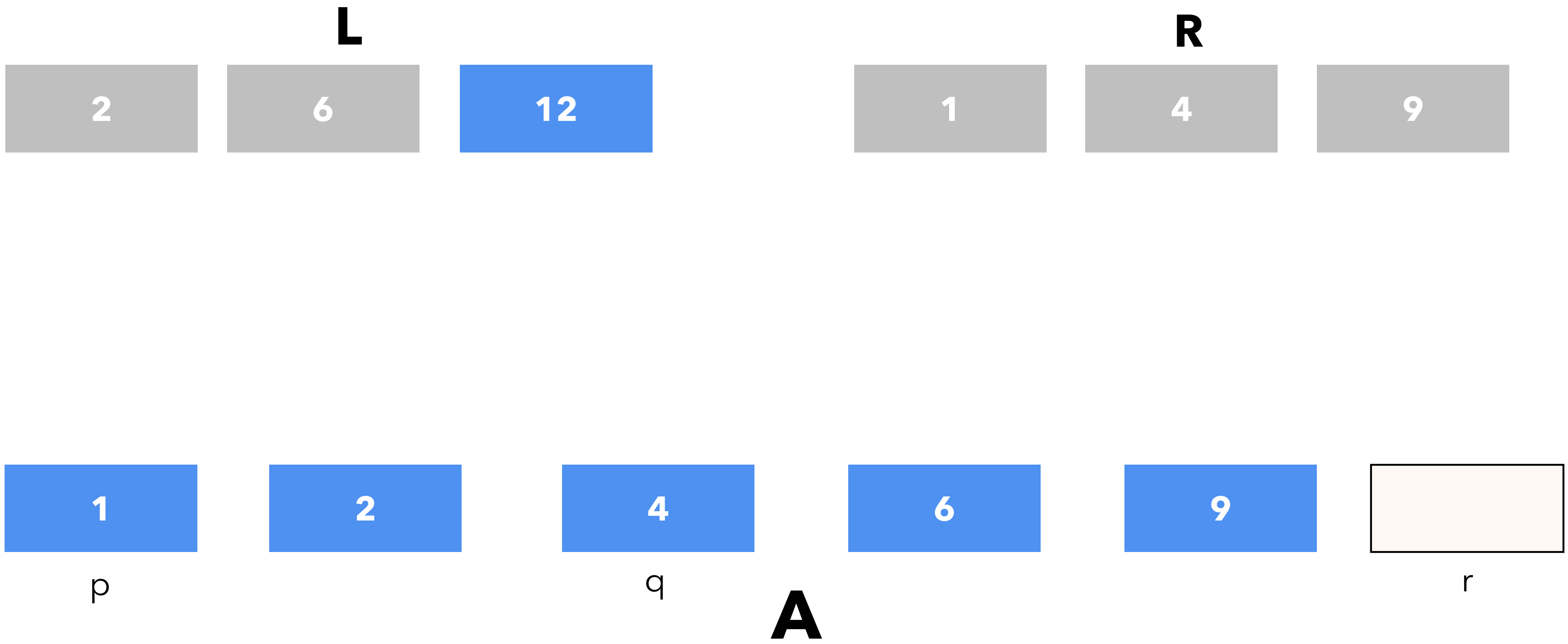
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

La procedura *merge* (esempio 1)



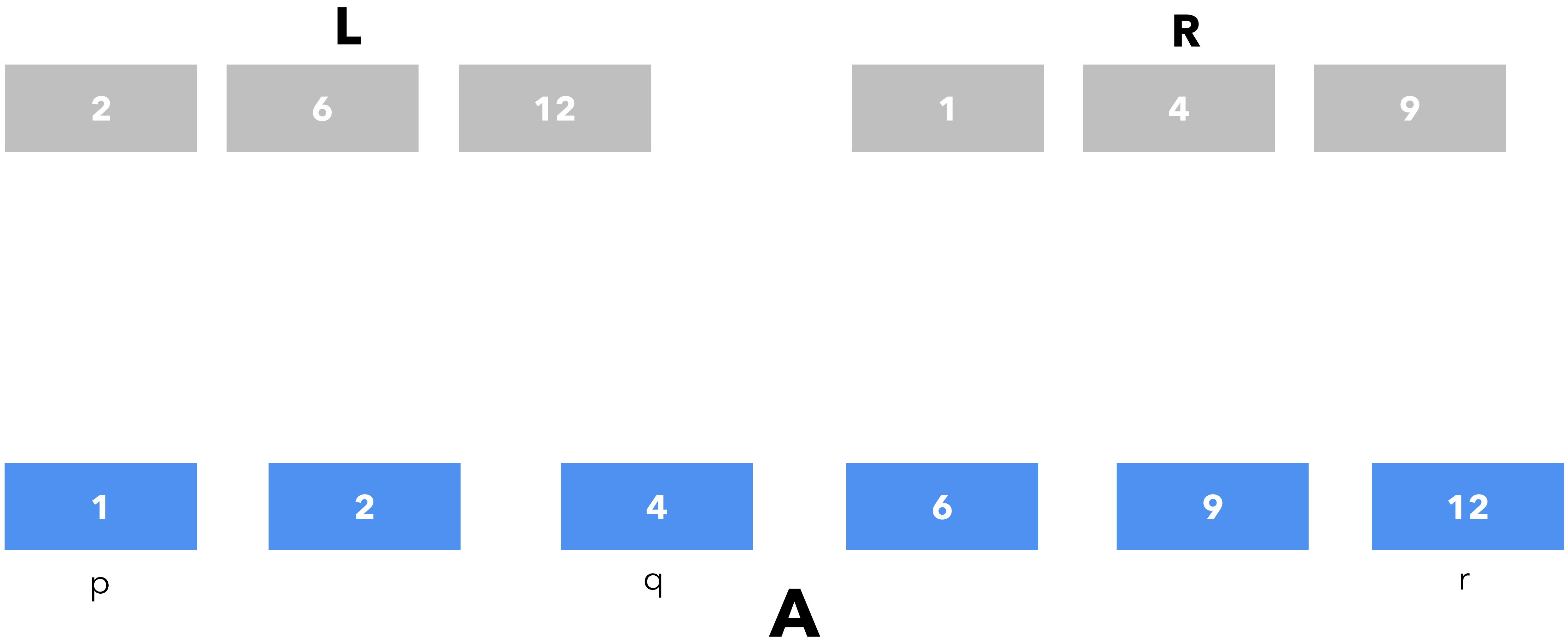
gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

La procedura *merge* (esempio 1)



gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

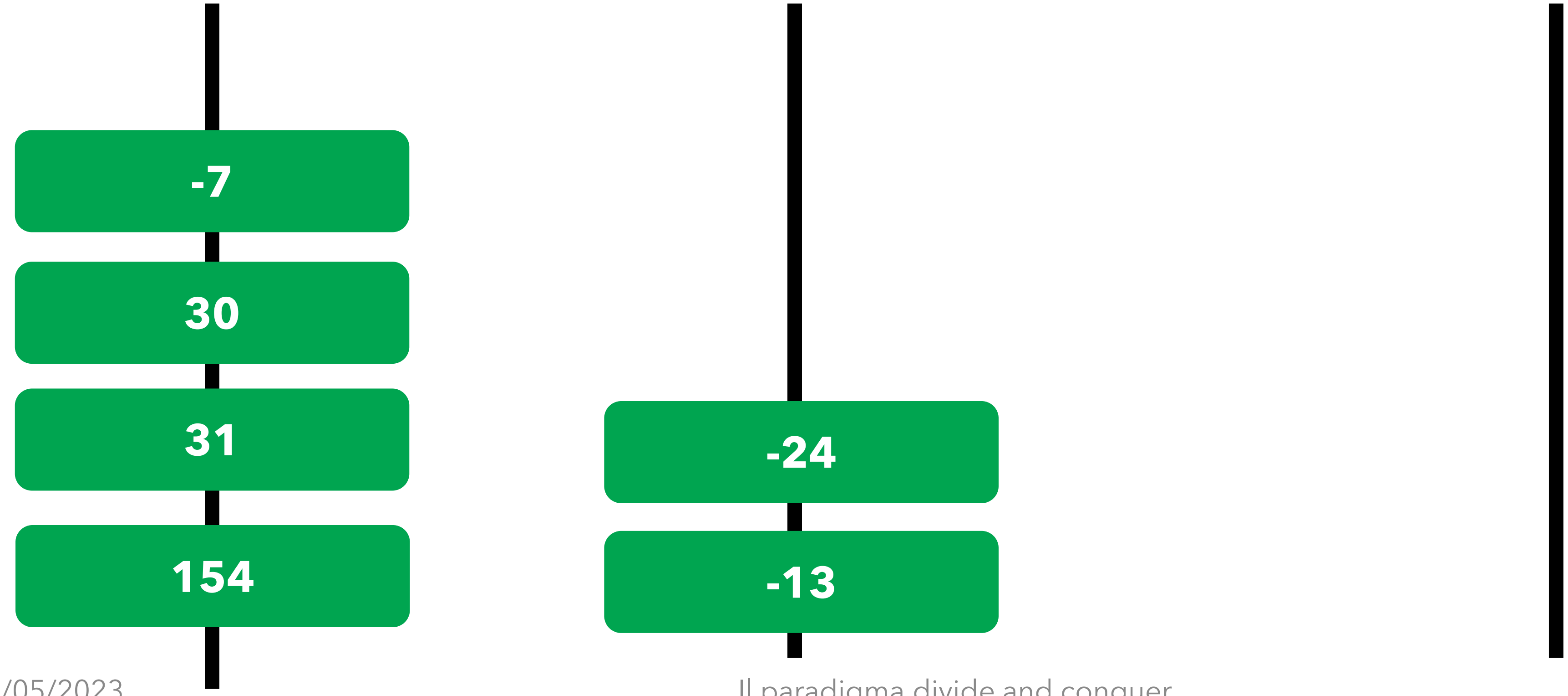
La procedura *merge* (esempio 1)



gli elementi colorati in blu vengono confrontati. Il più piccolo viene posto nell'array A.
In seguito non verrà più considerato, in quanto già «sistemato».
Gli elementi già sistemati sono colorati in grigio.

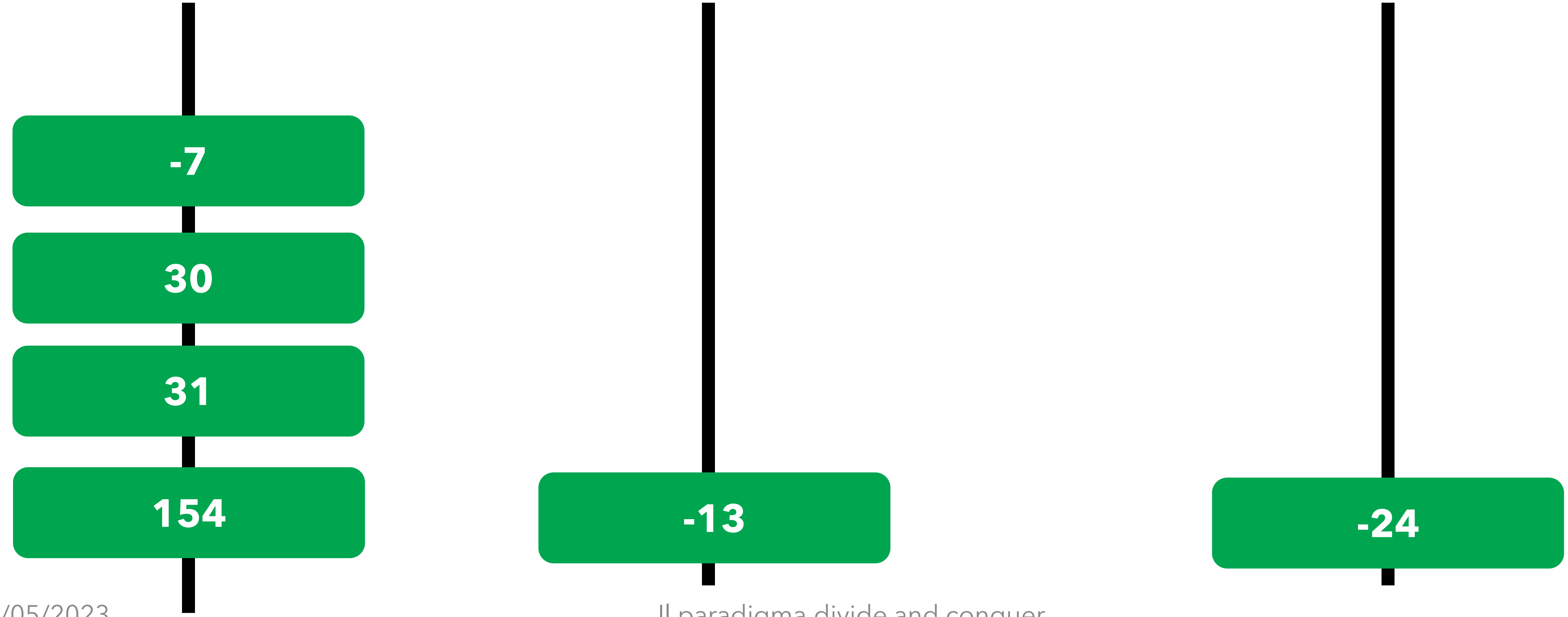
La procedura *merge*

- Per scrivere la procedura merge bisogna prestare molta attenzione a questo fatto: procedendo con la fusione dei due array, ad un certo punto ci si ritrova sempre con uno dei due array vuoto, ossia completamente *sistemato*
- Consideriamo un esempio in cui questo fatto è evidente. Mergiamo due mazzi di carte ordinati e impilati così. Confrontiamo sempre le carte in cima alle pile e impiliamo la più piccola nel palo vuoto:



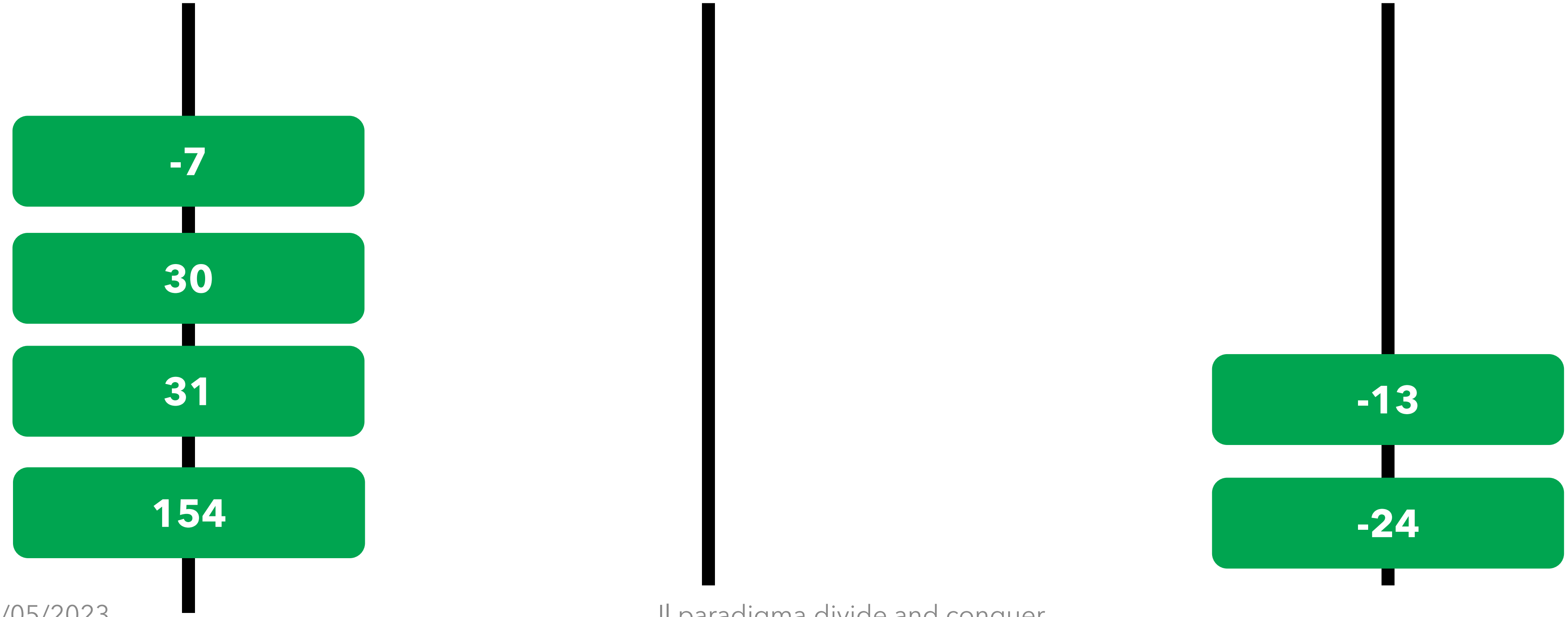
La procedura *merge*

- Per scrivere la procedura merge bisogna prestare molta attenzione a questo fatto: procedendo con la fusione dei due array, ad un certo punto ci si ritrova sempre con uno dei due array vuoto, ossia completamente *sistemato*
- Consideriamo un esempio in cui questo fatto è evidente. Mergiamo due mazzi di carte ordinati e impilati così. Confrontiamo sempre le carte in cima alle pile e impiliamo la più piccola nel palo vuoto:



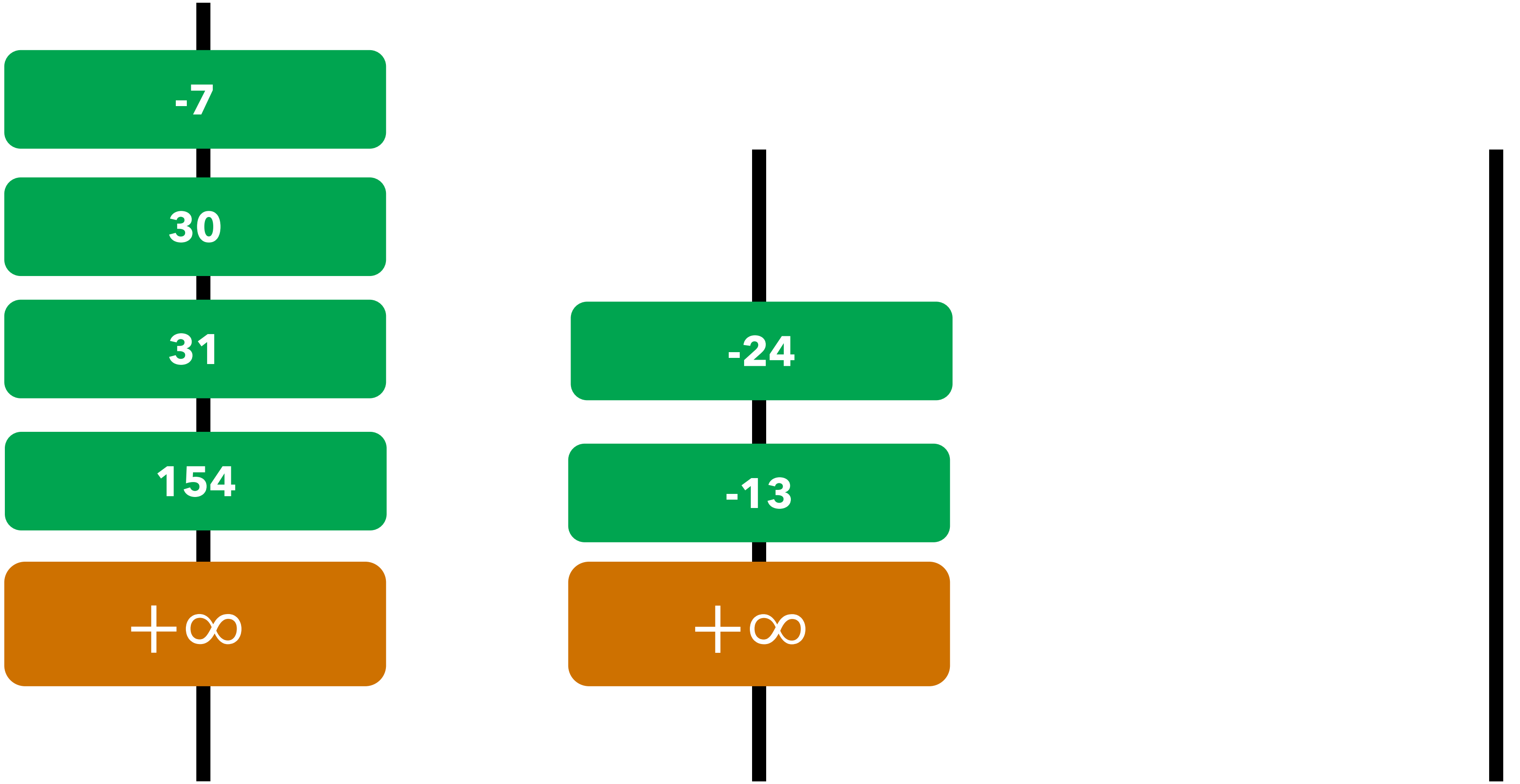
La procedura *merge*

- Ci siamo ritrovati con il secondo palo vuota!
- Dobbiamo semplicemente impilare tutti gli elementi della prima pila nella pila di output
- Ci serve un modo per capire subito che una pila si è svuotata completamente



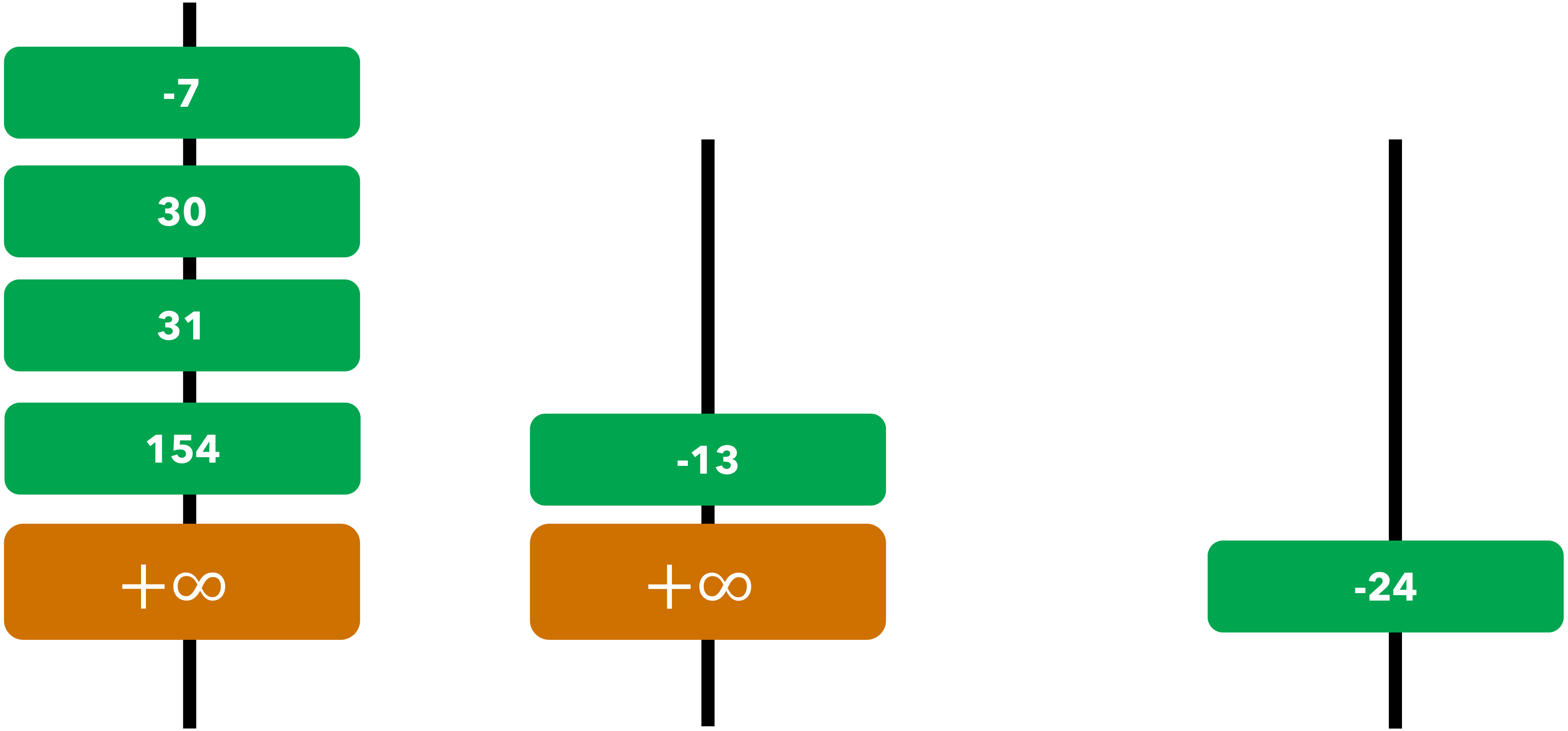
La procedura *merge*

- Poniamo dei un **valore sentinella** alla base delle pile: scegliamo come sentinella $+\infty$, perché per qualsiasi confronto tra un numero n e $+\infty$ risulta che n è minore, per cui sarà proprio n ad essere scelto



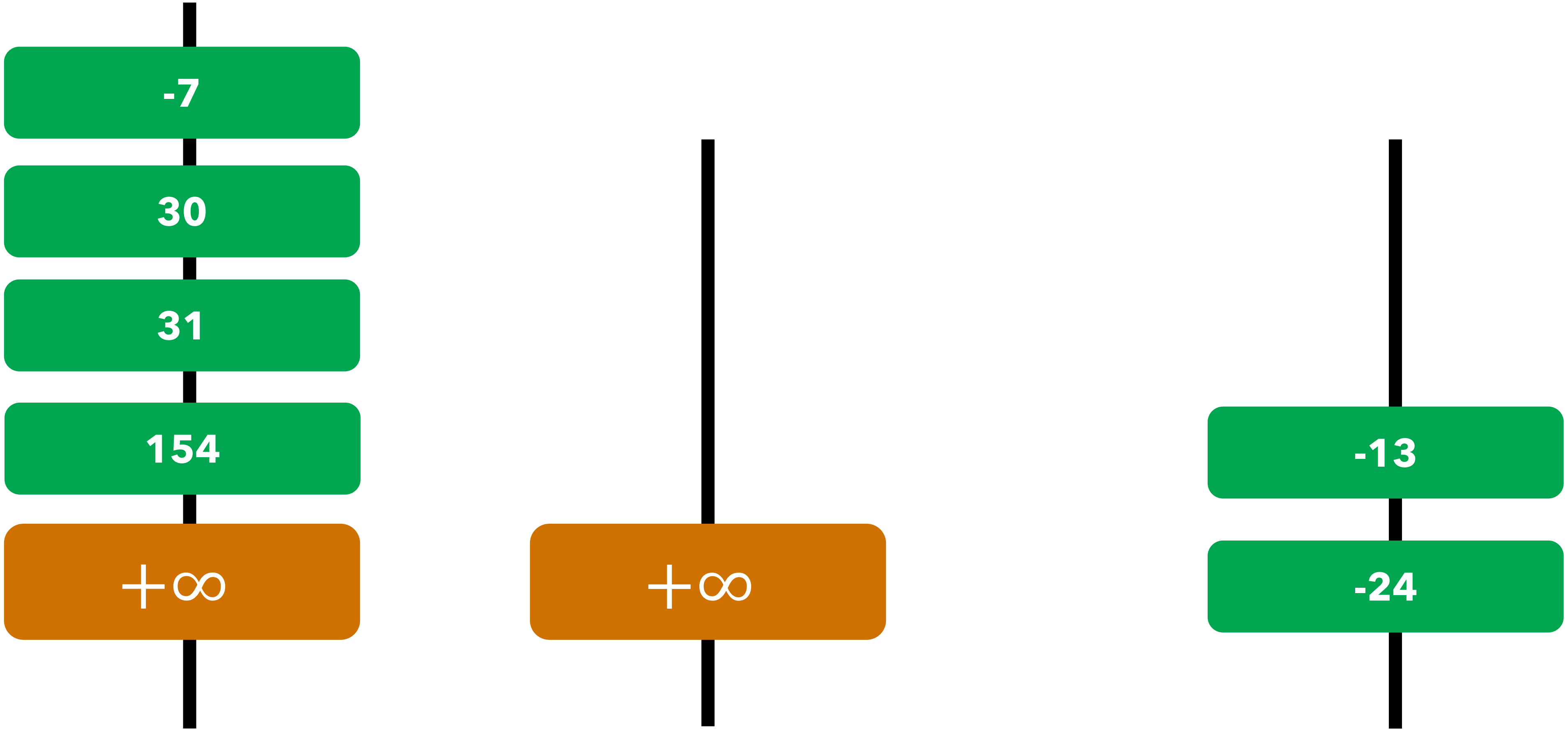
La procedura *merge*

- Poniamo dei un **valore sentinella** alla base delle pile: scegliamo come sentinella $+\infty$, perché per qualsiasi confronto tra un numero n e $+\infty$ risulta che n è minore, per cui sarà proprio n ad essere scelto



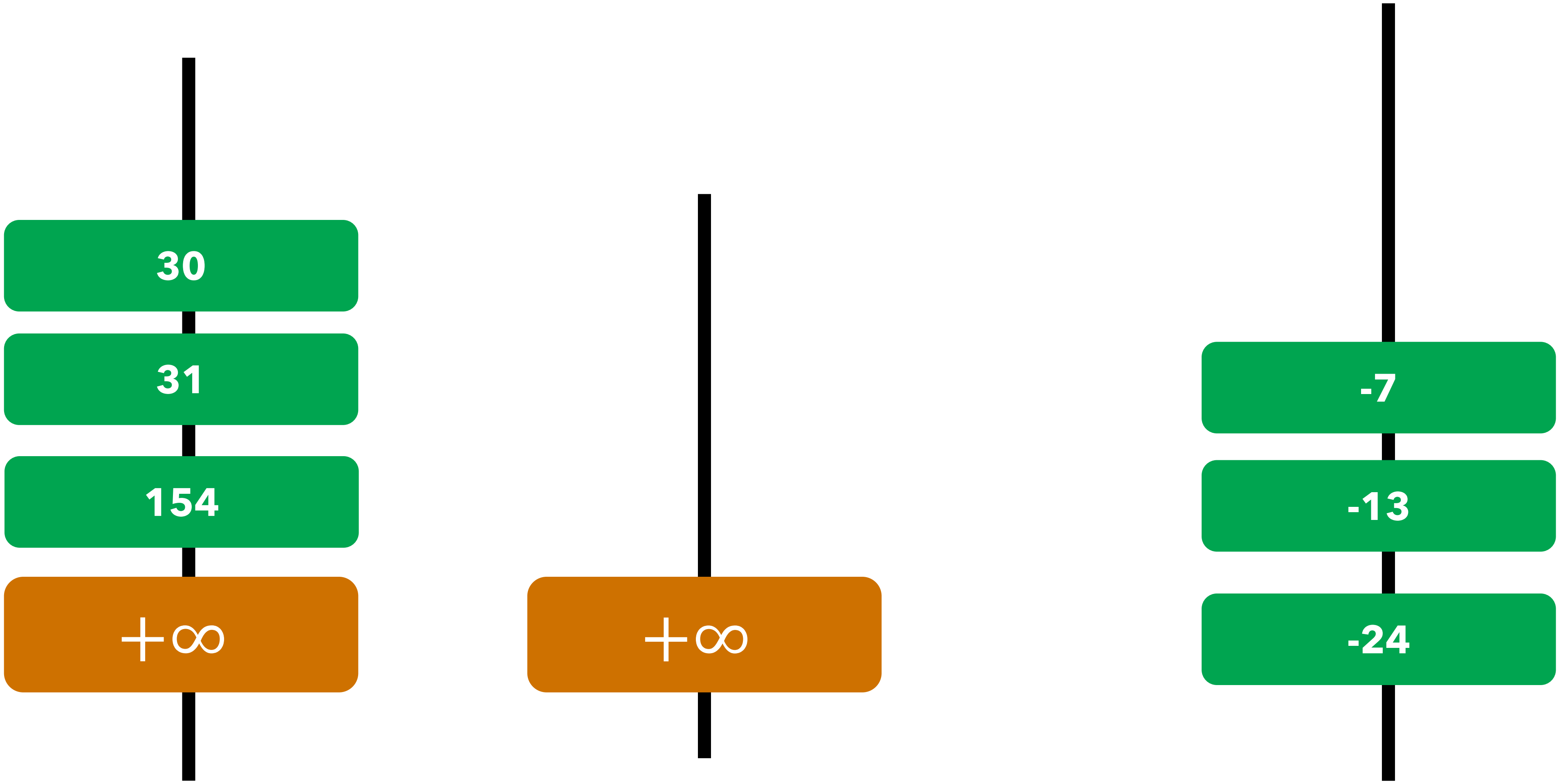
La procedura *merge*

- Confrontiamo -7 con +∞: ovviamente -7 <= +∞, quindi impiliamo –7 nella pila di output. Vedete che non abbiamo neanche controllato se la seconda pila era vuota. C'è semplicemente un valore comodo per mandare avanti la procedura!



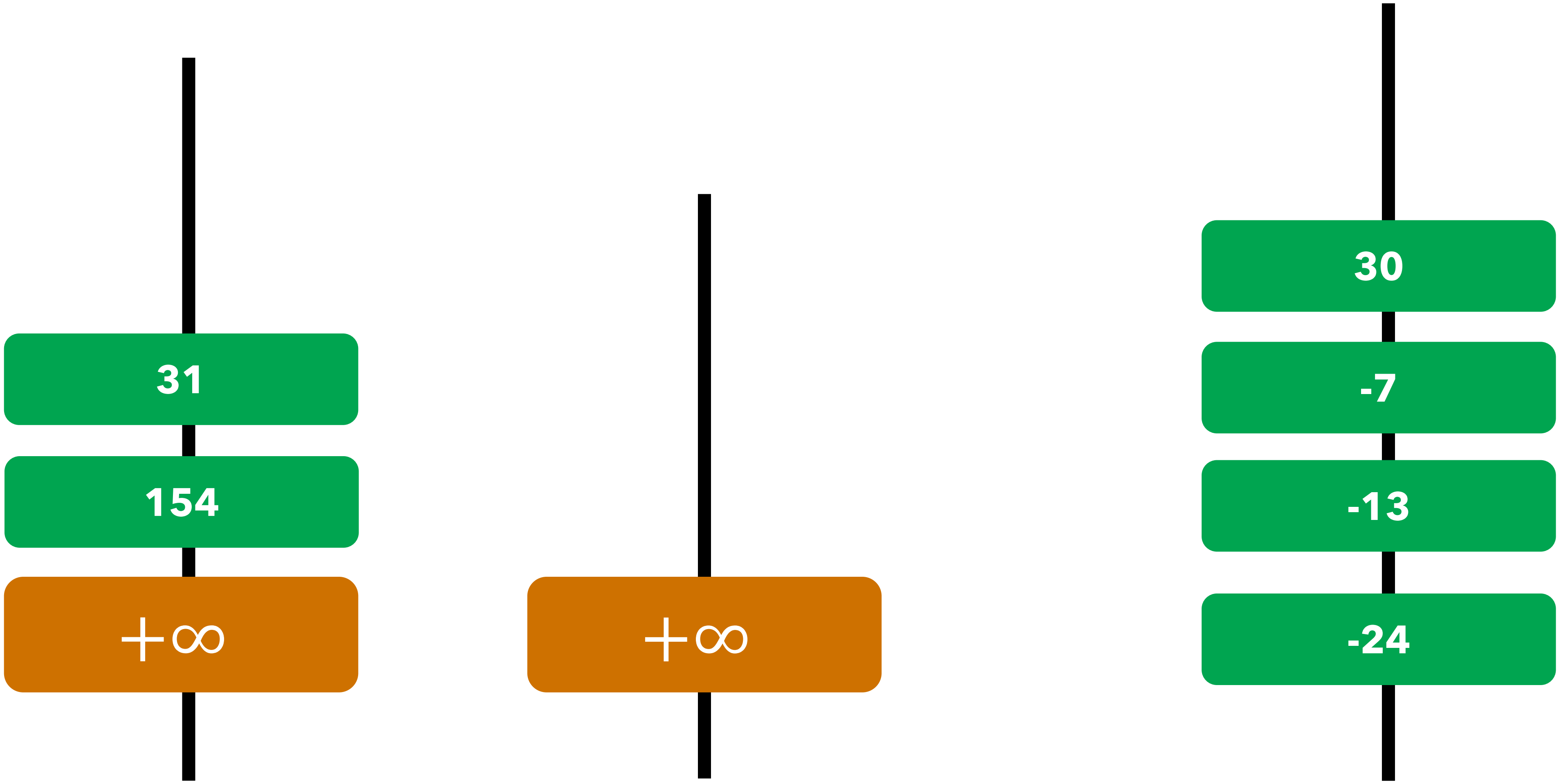
La procedura *merge*

- Confrontiamo -7 con +∞: ovviamente -7 <= +∞, quindi impiliamo –7 nella pila di output. Vedete che non abbiamo neanche controllato se la seconda pila era vuota. C'è semplicemente un valore comodo per mandare avanti la procedura!



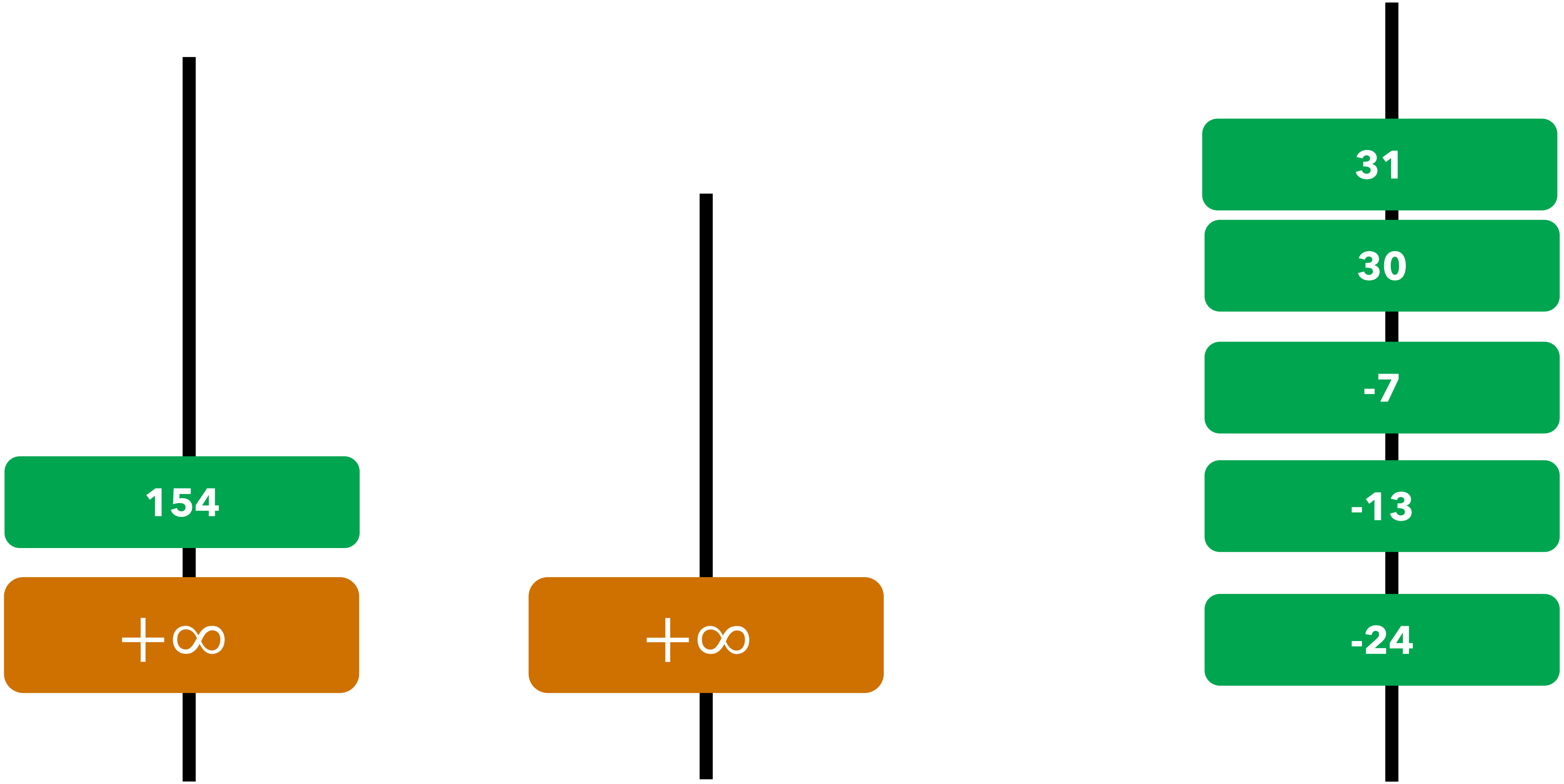
La procedura *merge*

- Confrontiamo -7 con +∞: ovviamente -7 <= +∞, quindi impiliamo –7 nella pila di output. Vedete che non abbiamo neanche controllato se la seconda pila era vuota. C'è semplicemente un valore comodo per mandare avanti la procedura!



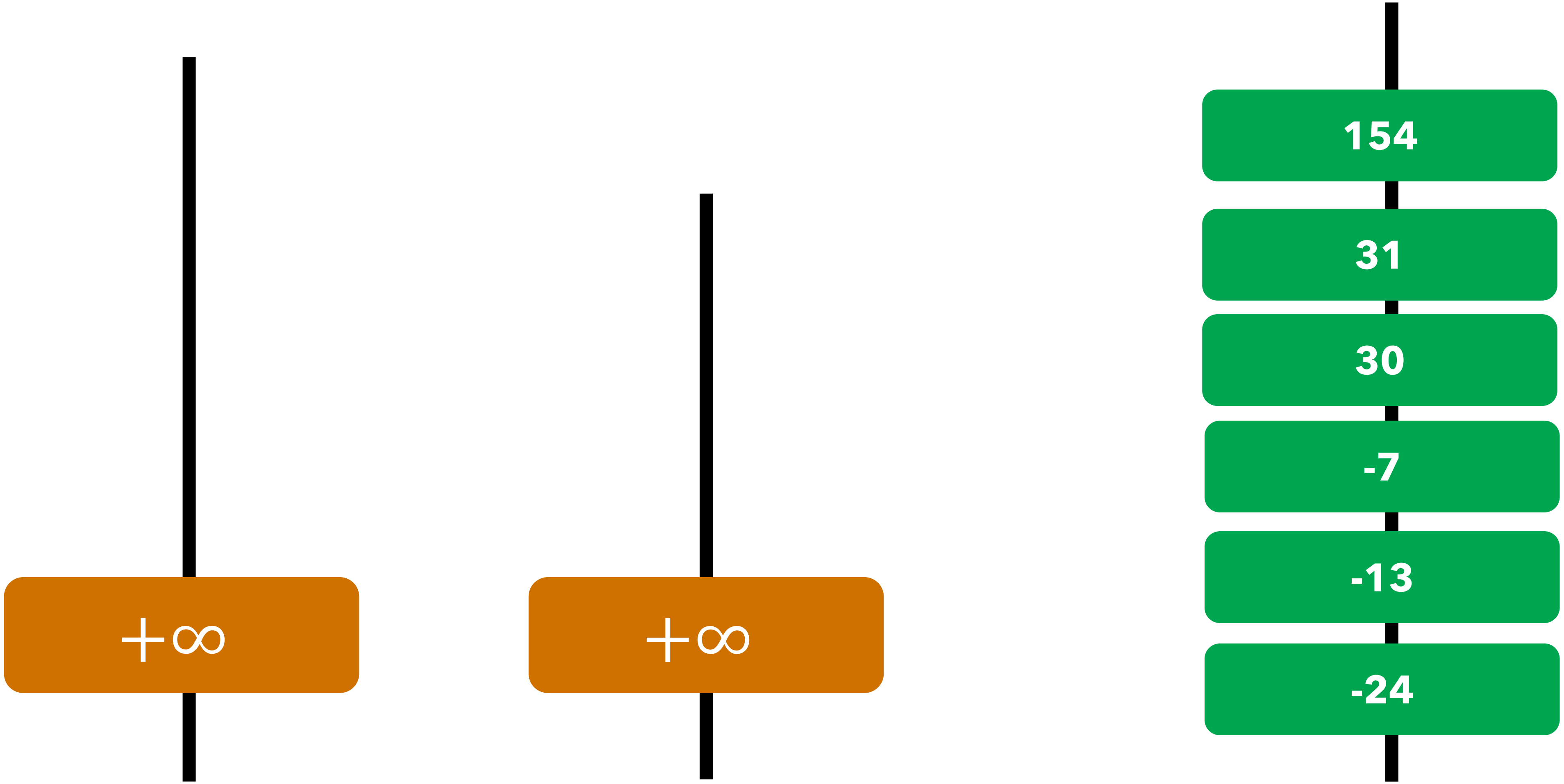
La procedura *merge*

- Confrontiamo -7 con +∞: ovviamente -7 <= +∞, quindi impiliamo –7 nella pila di output. Vedete che non abbiamo neanche controllato se la seconda pila era vuota. C'è semplicemente un valore comodo per mandare avanti la procedura!



La procedura *merge*

- Confrontiamo -7 con +∞: ovviamente -7 <= +∞, quindi impiliamo –7 nella pila di output. Vedete che non abbiamo neanche controllato se la seconda pila era vuota. C'è semplicemente un valore comodo per mandare avanti la procedura!



La procedura *merge* - pseudocodice

```
merge(A, low, middle, high):
    #needs additional memory, this is not an in-place algorithm
    left = A[low .. middle]
    right = A[middle + 1 .. high]
    left.add(+INF)
    right.add(+INF)

    left_i = 0
    right_i = 0

    for i from low to high:
        if left[left_i] <= right[right_i]:
            a[i] = left[left_i]
            left_i = left_i + 1
        else:
            a[i] = right[right_i]
            right_i = right_i + 1
```

Merge Sort, informalmente

- Sfruttiamo la procedura *merge* per ordinare un array!
- Vediamo il problema dell'ordinamento ricorsivamente:
  - Se l'array A ha dimensione > 1:
    - ordiniamo la metà inferiore di A (fase *divide*)
    - ordiniamo la metà superiore di A (fase *divide*)
    - fondiamo le 2 metà ordinate, utilizzando *merge* (fase *conquer*)
  - Se l'array A ha dimensione <= 1:
    - non serve fare niente. Un array vuoto, o di un solo elemento, è banalmente ordinato

Merge Sort, informalmente

12	9	18	-13	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente

12	9	18	-13	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente

12	9	18	-13	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente

12	9	18	-13	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

12	9	18	-13	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

12	9	18	-13	100	40	-1	-3	9	22

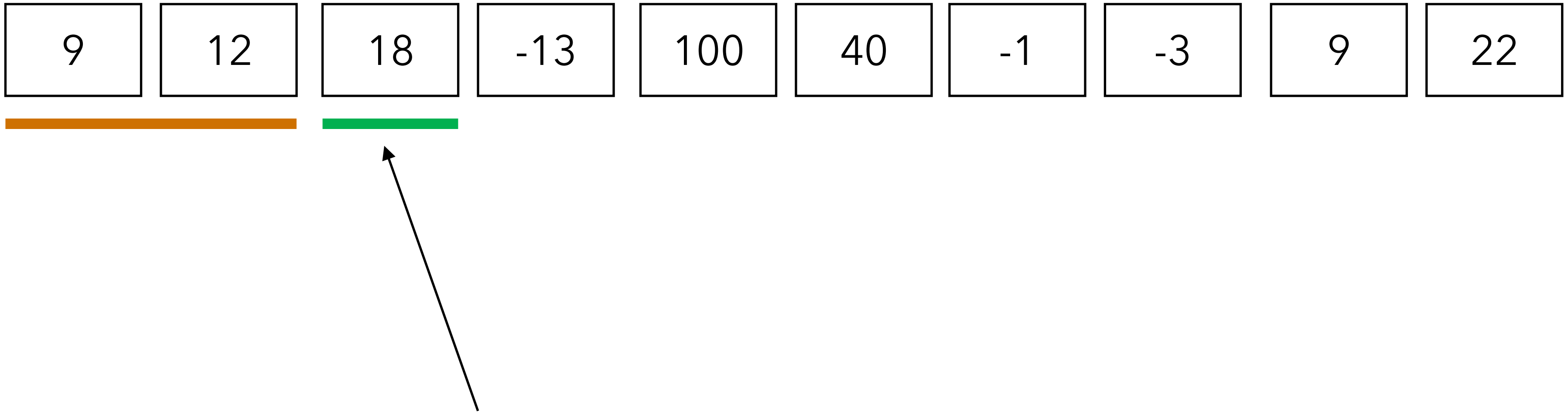
applichiamo *merge* sulla parte marrone

Merge Sort, informalmente

9	12	18	-13	100	40	-1	-3	9	22

la parte marrone è ordinata

Merge Sort, informalmente



applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

9	12	18	-13	100	40	-1	-3	9	22

applichiamo *merge* sulla parte marrone

Merge Sort, informalmente

9	12	18	-13	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente

9	12	18	-13	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

9	12	18	-13	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

9	12	18	-13	100	40	-1	-3	9	22

applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente

9	12	18	-13	100	40	-1	-3	9	22

applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente

-13	9	12	18	100	40	-1	-3	9	22

la parte marrone ora risulta ordinata

Merge Sort, informalmente

-13	9	12	18	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente

-13	9	12	18	100	40	-1	-3	9	22

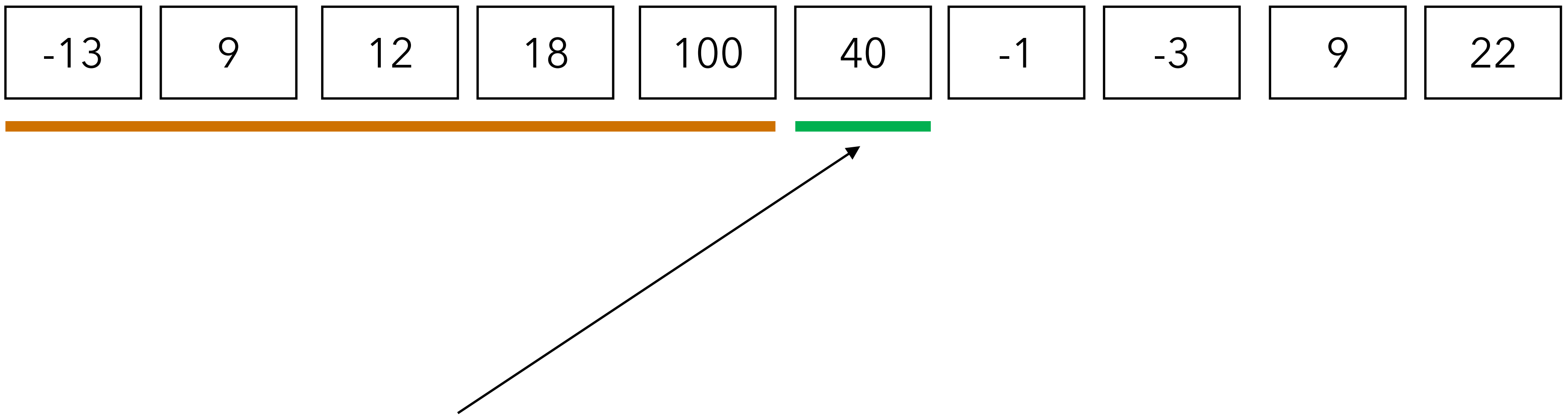
applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente

-13	9	12	18	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente



applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

-13	9	12	18	100	40	-1	-3	9	22

applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

-13	9	12	18	100	40	-1	-3	9	22

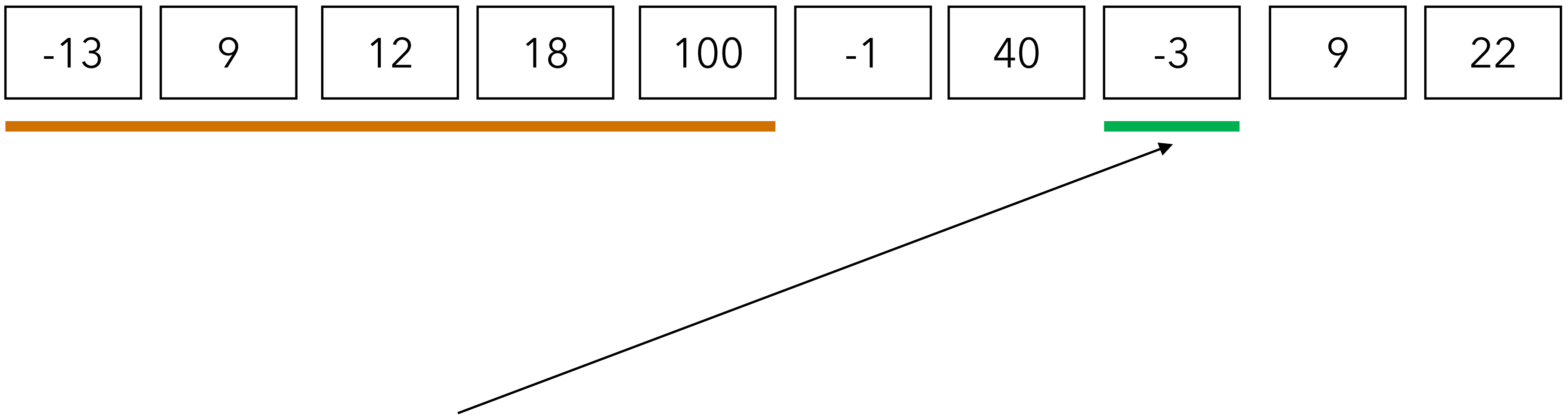
applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente

-13	9	12	18	100	-1	40	-3	9	22

applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente



applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

-13	9	12	18	100	-1	40	-3	9	22

applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente

-13	9	12	18	100	-3	-1	40	9	22

merge sistemato la fetta indicata

Merge Sort, informalmente

-13	9	12	18	100	-3	-1	40	9	22

applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente

-13	9	12	18	100	-3	-1	40	9	22

applichiamo *Merge Sort* sulla parte verde
→ 1 solo elemento, niente da fare

Merge Sort, informalmente

-13	9	12	18	100	-3	-1	40	9	22

applichiamo *Merge Sort* sulla parte verde

Merge Sort, informalmente

-13	9	12	18	100	-3	-1	40	9	22

applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente

-13	9	12	18	100	-3	-1	40	9	22

applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente

-13	9	12	18	100	-3	-1	40	9	22

applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente

-13	9	12	18	100	-3	-1	9	22	40

applichiamo *merge* sulla parte marrone indicata

Merge Sort, informalmente

-13	-3	-1	9	9	12	18	22	40	100

merge è stato applicato sulle due metà dell'array iniziale → l'array è stato ordinato

Merge Sort - pseudocodice

```
merge_sort(A, low, high):
    if low >= high:
        return
    middle = (low + high) / 2
    merge_sort(A, low, middle)
    merge_sort(A, middle + 1, high)
    merge(A, low, middle, high)
```

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**18**	**9**	**5**	**18**	**12**	**4**	**101**
	low			middle			high

C A L L S T A C K

low: 0   high: 6   middle: 3

external caller's call

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**18**	**9**	**5**	**18**	**12**	**4**	**101**
	low	middle		high			

C A L L S T A C K

low: 0 high: 3 middle: 1	*1st (aka left) recursive call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**18**	**9**	**5**	**18**	**12**	**4**	**101**
	low middle	high					

CALL STACK

low: 0 high: 1 middle: 0 — *1st (aka left) recursive call*

low: 0 high: 3 middle: 1 — *1st (aka left) recursive call*

low: 0 high: 6 middle: 3 — *external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**18**	**9**	**5**	**18**	**12**	**4**	**101**
	low middle high						

C A L L S T A C K

low: 0 high: 0 middle: 0	→ *1st (aka left) recursive call, base case → return and pop the stack frame*
low: 0 high: 1 middle: 0	*1st (aka left) recursive call*
low: 0 high: 3 middle: 1	*1st (aka left) recursive call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**18**	**9**	**5**	**18**	**12**	**4**	**101**
	low middle	high					

C A L L S T A C K

low: 0 high: 1 middle: 0	*1st (aka left) recursive call, just returned from left call, make 2nd call*
low: 0 high: 3 middle: 1	*1st (aka left) recursive call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**18**	**9**	**5**	**18**	**12**	**4**	**101**
		low middle high					

CALL STACK

Stack frame	
low: 1 high: 1 middle: 1	→ *2nd (aka right) recursive call* *base case → return and pop the stack frame*
low: 0 high: 1 middle: 0	*1st (aka left) recursive call*
low: 0 high: 3 middle: 1	*1st (aka left) recursive call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**18**	**9**	**5**	**18**	**12**	**4**	**101**
	low middle	high					

C A L L S T A C K

low: 0 high: 1 middle: 0	*1st (aka left) recursive call, just returned from right call, call merge(0, 0, 1)*
low: 0 high: 3 middle: 1	*1st (aka left) recursive call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**9**	**18**	**5**	**18**	**12**	**4**	**101**
	low middle	high					

C A L L S T A C K

low: 0 high: 1 middle: 0 → *1st (aka left) recursive call called merge(0, 0, 1) A[0:1] is sorted, nothing to do after merge, pop the stack frame*

low: 0 high: 3 middle: 1 *1st (aka left) recursive call*

low: 0 high: 6 middle: 3 *external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**9**	**18**	**5**	**18**	**12**	**4**	**101**
	low	middle		high			

C A L L S T A C K

low: 0 high: 3 middle: 1 — *1st (aka left) recursive call, just returned from left call, make 2nd call*

low: 0 high: 6 middle: 3 — *external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**9**	**18**	**5**	**18**	**12**	**4**	**101**
			low middle	high			

CALL STACK

low: 2 high: 3 middle: 2	*2nd (aka right) recursive call*
low: 0 high: 3 middle: 1	*1st (aka left) recursive call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**9**	**18**	**5**	**18**	**12**	**4**	**101**
			low				
			middle				
			high				

CALL STACK

low: 2 high: 2 middle: 2	*1st (aka left) recursive call, base case → return and pop*
low: 2 high: 3 middle: 2	*2nd (aka right) recursive call*
low: 0 high: 3 middle: 1	*1st (aka left) recursive call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul call stack

	0	1	2	3	4	5	6
A	**9**	**18**	**5**	**18**	**12**	**4**	**101**
			low	high			
			middle				

C A L L S T A C K

low: 2 high: 3 middle: 2	*2nd (aka right) recursive call, just returned from left call, make 2nd call*
low: 0 high: 3 middle: 1	*1st (aka left) recursive call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**9**	**18**	**5**	**18**	**12**	**4**	**101**
				low middle high			

C A L L S T A C K

low: 3 high: 3 middle: 3 — *2nd (aka right) recursive call, base case, return and pop*

low: 2 high: 3 middle: 2 — *2nd (aka right) recursive call*

low: 0 high: 3 middle: 1 — *1st (aka left) recursive call*

low: 0 high: 6 middle: 3 — *external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**9**	**18**	**5**	**18**	**12**	**4**	**101**
			low middle	high			

CALL STACK

low: 2 high: 3 middle: 2 — *2nd (aka right) recursive call, just returned from right call → call merge(2, 2, 3), return and pop*

low: 0 high: 3 middle: 1 — *1st (aka left) recursive call*

low: 0 high: 6 middle: 3 — *external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**9**	**18**	**5**	**18**	**12**	**4**	**101**
	low	middle		high			

C
A
L
L

S
T
A
C
K

Call stack	
low: 0 high: 3 middle: 1	*1st (aka left) recursive call, just returned from right call, call merge(0, 1, 3)*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**5**	**9**	**18**	**18**	**12**	**4**	**101**
	low	middle		high			

C A L L S T A C K

low: 0 high: 3 middle: 1

A[0:3] is sorted, nothing to do after merge, return and pop

low: 0 high: 6 middle: 3

external caller's call

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**5**	**9**	**18**	**18**	**12**	**4**	**101**
	low			middle			high

C A L L S T A C K

low: 0   high: 6   middle: 3

external caller's call, just returned from left call, make right call

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**5**	**9**	**18**	**18**	**12**	**4**	**101**
					low	middle	high

C A L L S T A C K

low: 4 high: 6 middle: 5	*2nd (aka right) call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**5**	**9**	**18**	**18**	**12**	**4**	**101**
					low middle	high	

C A L L S T A C K

low: 4 high: 5 middle: 4	*1st (aka left call)*
low: 4 high: 6 middle: 5	*2nd (aka right) call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**5**	**9**	**18**	**18**	**12**	**4**	**101**
					low middle high		

CALL STACK

low: 4 high: 4 middle: 4	*1st (aka left call), base case → return and pop*
low: 4 high: 5 middle: 4	*1st (aka left call)*
low: 4 high: 6 middle: 5	*2nd (aka right) call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**5**	**9**	**18**	**18**	**12**	**4**	**101**
					low middle	high	

C A L L S T A C K

Call stack	
low: 4 high: 5 middle: 4	*1st (aka left call), just returned from left call, make 2nd call*
low: 4 high: 6 middle: 5	*2nd (aka right) call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**5**	**9**	**18**	**18**	**12**	**4**	**101**
						low middle high	

C A L L S T A C K

Stack	
low: 5 high: 5 middle: 5	*2nd (aka right) call, base case, return and pop*
low: 4 high: 5 middle: 4	*1st (aka left call)*
low: 4 high: 6 middle: 5	*2nd (aka right) call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**5**	**9**	**18**	**18**	**12**	**4**	**101**
					low middle	high	

C A L L S T A C K

low: 4 high: 5 middle: 4 — *1st (aka left call), just returned from right call, call merge(4, 4, 5)*

low: 4 high: 6 middle: 5 — *2nd (aka right) call*

low: 0 high: 6 middle: 3 — *external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**5**	**9**	**18**	**18**	**4**	**12**	**101**
					low middle	high	

C A L L S T A C K

low: 4 high: 5 middle: 4	*1st (aka left) call,* *A[4:5] is sorted, nothing to do after merge, return and pop*
low: 4 high: 6 middle: 5	*2nd (aka right) call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**5**	**9**	**18**	**18**	**4**	**12**	**101**
					low	middle	high

CALL STACK

Stack frame	
low: 4 high: 6 middle: 5	*2nd (aka right) call, just returned from left call, make 2nd call*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**5**	**9**	**18**	**18**	**4**	**12**	**101**
							low middle high

C A L L S T A C K

low: 6 high: 6 middle: 6 — *2nd (aka right) call, base case → return and pop*

low: 4 high: 6 middle: 5 — *2nd (aka right) call*

low: 0 high: 6 middle: 3 — *external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**5**	**9**	**18**	**18**	**4**	**12**	**101**
					low	middle	high

C A L L S T A C K

Call stack	
low: 4 high: 6 middle: 5	*2nd (aka right) call, just returned from right call, call merge(4, 5, 6)*
low: 0 high: 6 middle: 3	*external caller's call*

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**5**	**9**	**18**	**18**	**4**	**12**	**101**
					low	middle	high

C A L L S T A C K

low: 4 high: 6 middle: 5
low: 0 high: 6 middle: 3

2nd (aka right) call,
A[4:6] is sorted, nothing to do after merge,
return and pop

external caller's call, just returned from left call

Merge Sort, cosa succede sul *call stack*

	0	*1*	*2*	*3*	*4*	*5*	*6*
A	**5**	**9**	**18**	**18**	**4**	**12**	**101**
	low			middle			high

C A L L S T A C K

low: 0 high: 6 middle: 3

external caller's call, just returned from right call, call merge(0, 3, 6)

Merge Sort, cosa succede sul *call stack*

	0	1	2	3	4	5	6
A	**4**	**5**	**9**	**12**	**18**	**18**	**101**
	low			middle			high

C A L L S T A C K

low: 0 high: 6 middle: 3

nothing to do after merge, the array is completely sorted, return to external caller