Paradigma *divide et impera*
Recursive binary search
Merge sort

Liceo G.B. Brocchi – Bassano del Grappa (VI)
Liceo Scientifico – opzione scienze applicate
Giovanni Mazzocchin

Divide, conquer and combine

- Molti problemi sono risolvibili «naturalmente» in modo ricorsivo. Con *naturalmente* intendiamo 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 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, conquer and combine* si articola in 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
- 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

```
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

La ricerca binaria ricorsiva

```
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
```

La ricerca binaria ricorsiva

```
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_R(A, low, middle - 1, key)

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

Merge sort

- 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
    - **conquer**: ordina le 2 sottoliste ricorsivamente
    - **combine**: fondi (*merge*) le 2 sottoliste ordinate

- Merge sort è stato inventato da John von Neumann

La procedura *merge*

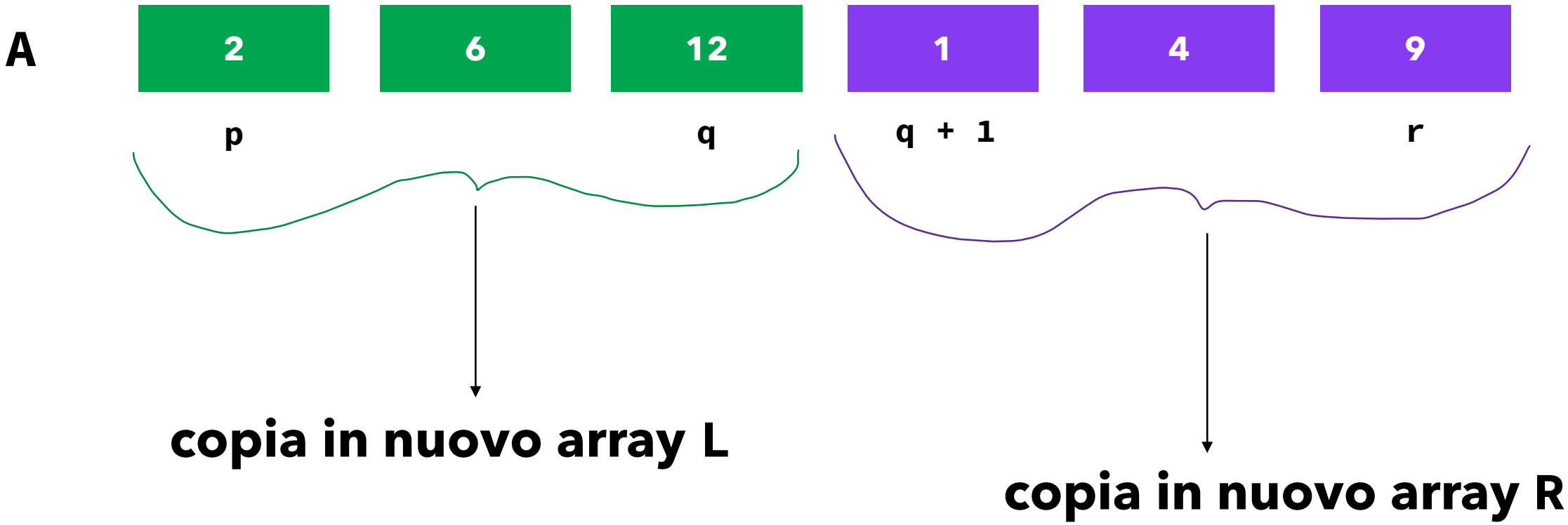
- Prima di scrivere il codice di Merge sort, abbiamo bisogno di definire la procedura **`merge(A, p, q, r)`**, i cui requisiti sono i seguenti:

`dato un array A e tre indici p, q, r, con 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] completamente ordinato, fondendo gli elementi di A[p..q] e A[q + 1, r]`

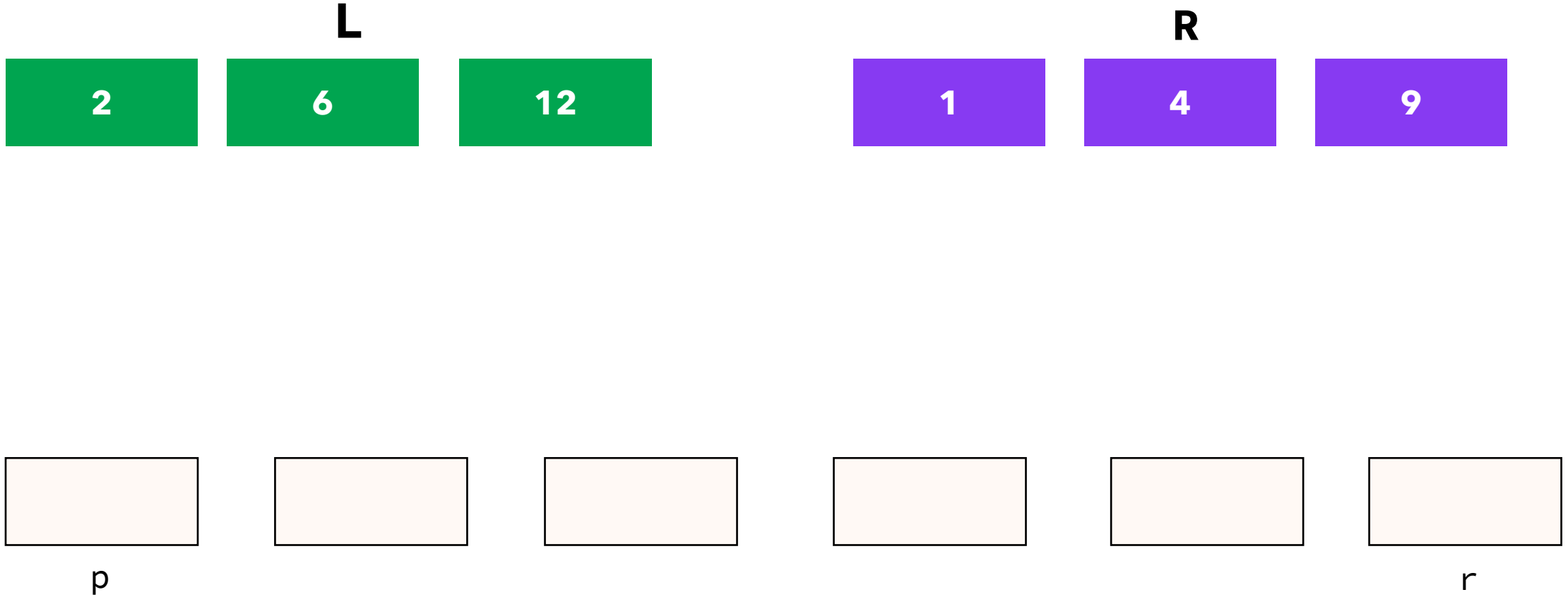
La procedura *merge*



mergiamo L e R nell'array A

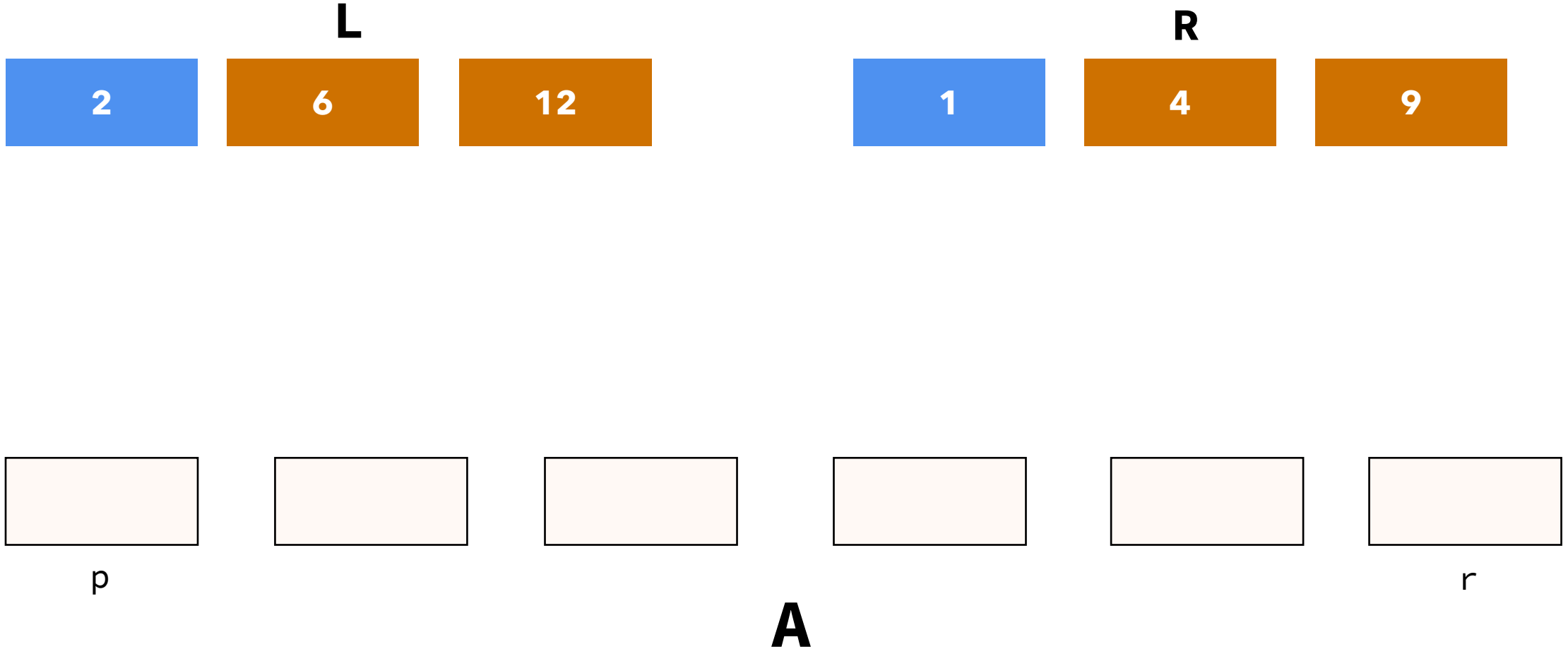
NB: numero degli elementi da mergiare è: `r - p + 1`

La procedura *merge*



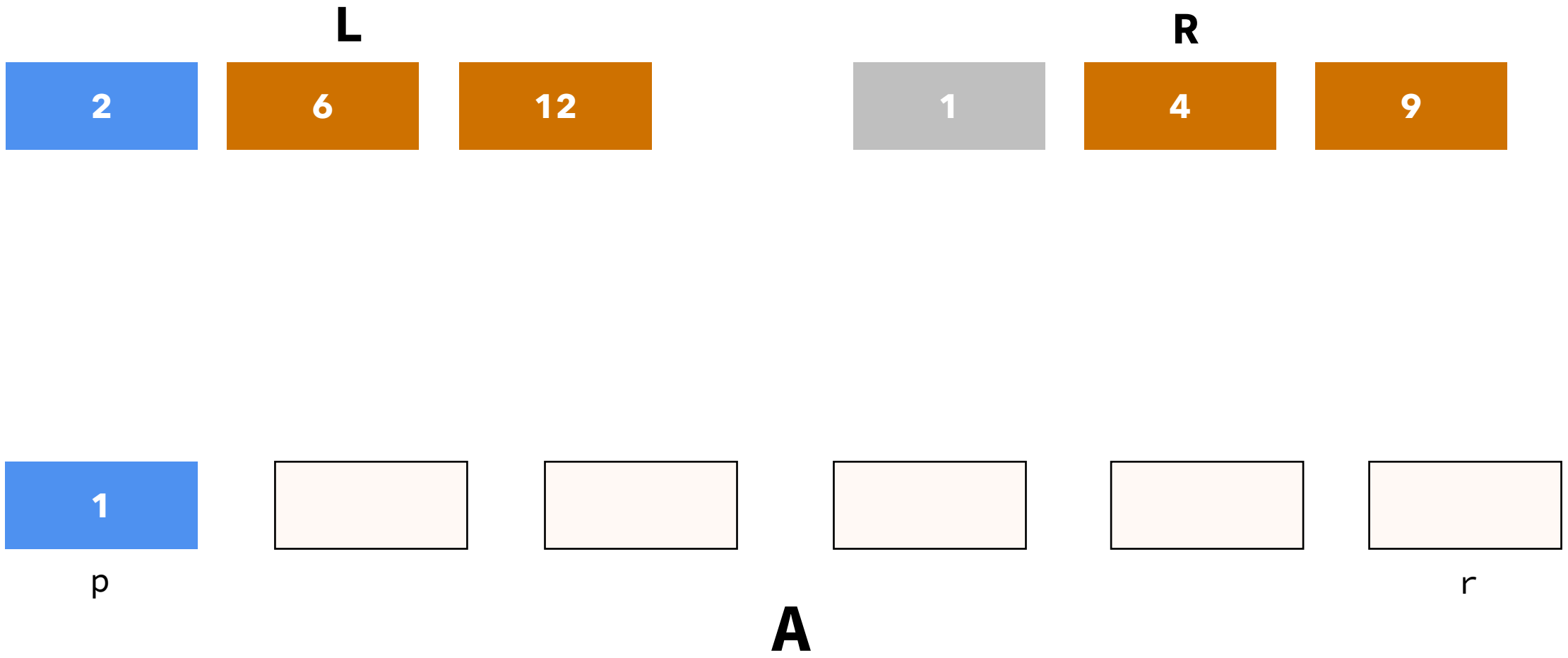
A va riempito da p a r con gli array L e R mergiati

La procedura *merge*

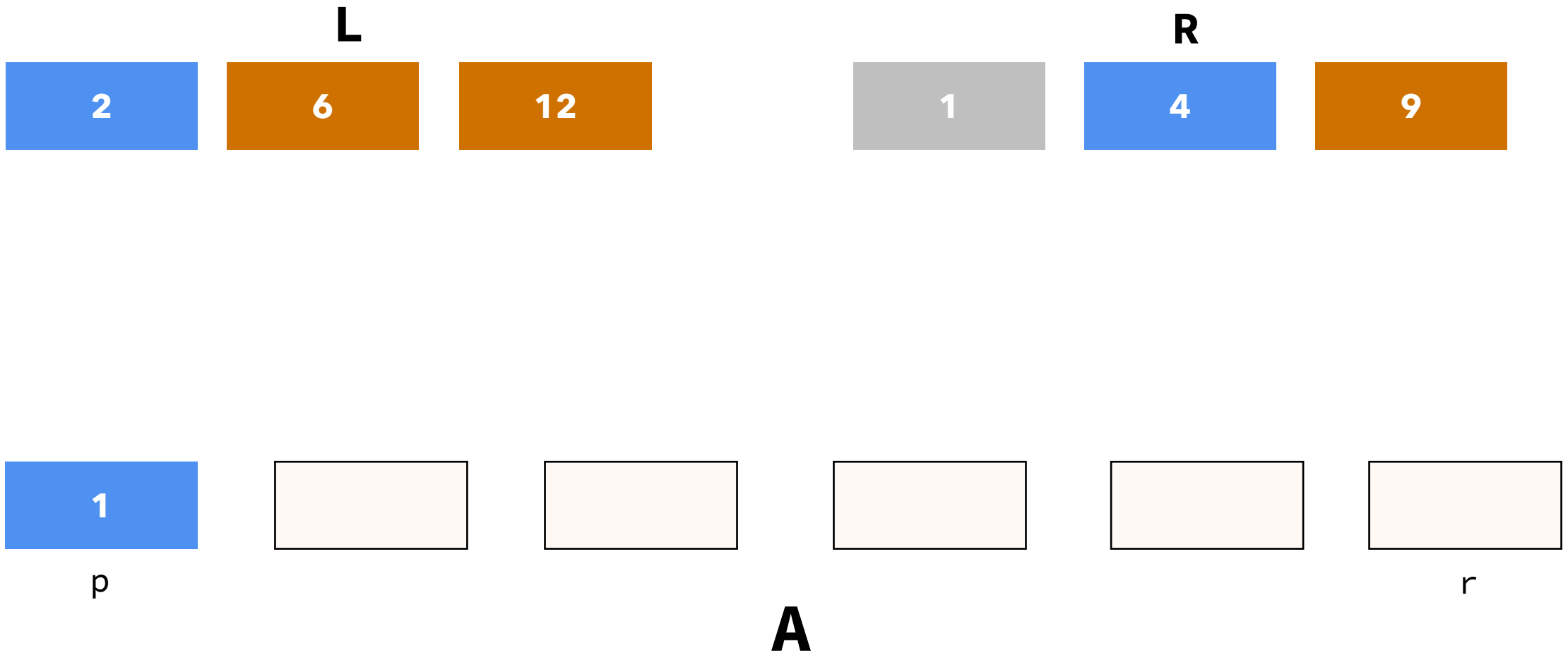


gli elementi blu vengono confrontati. Il più piccolo viene posto nell'array A. In seguito non verrà più considerato, in quanto già «sistemato», e diventerà grigio

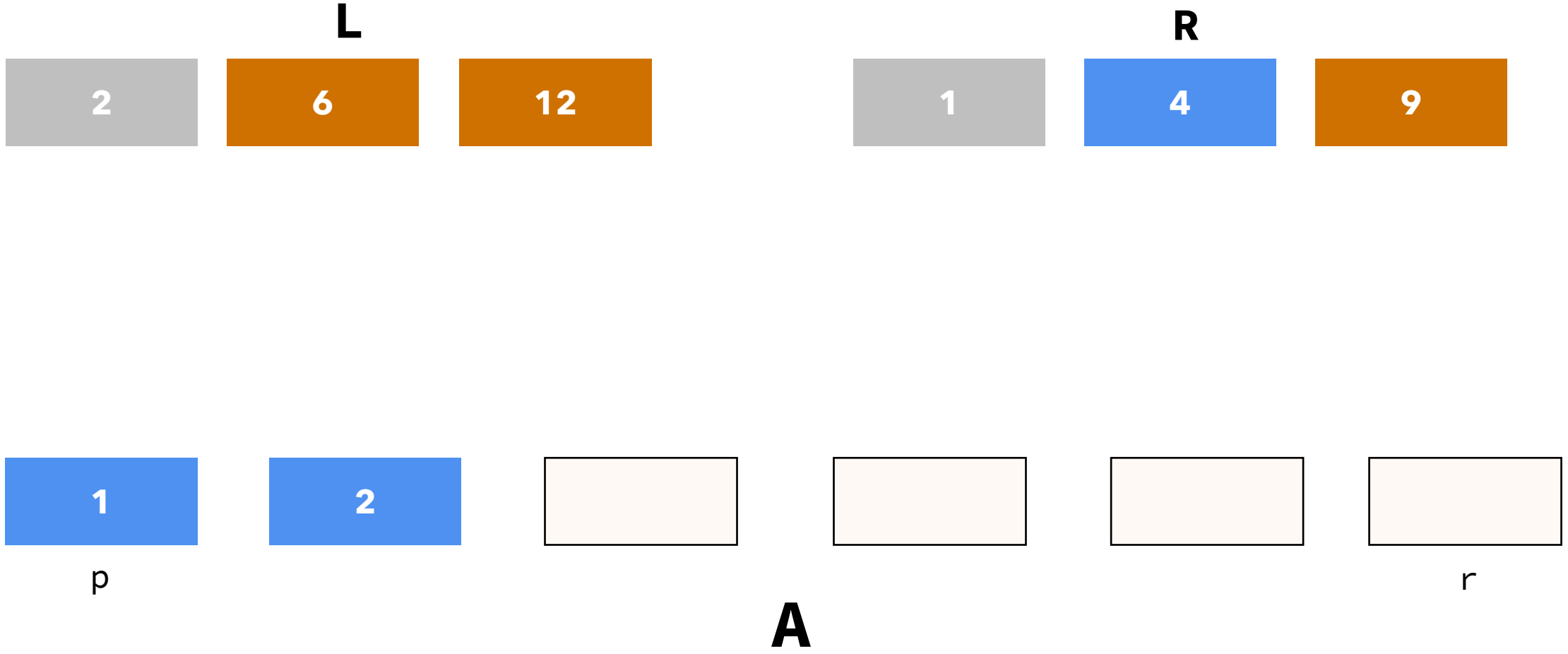
La procedura *merge*



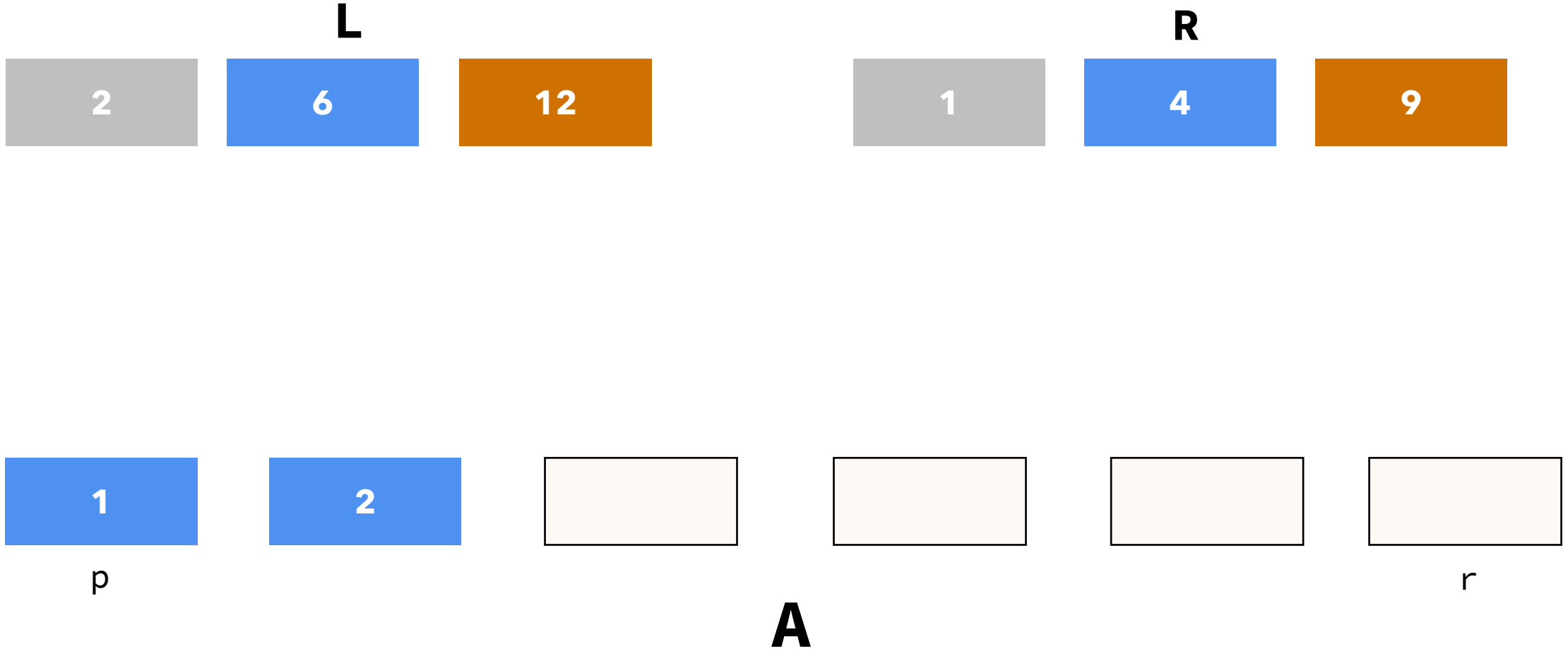
La procedura *merge*



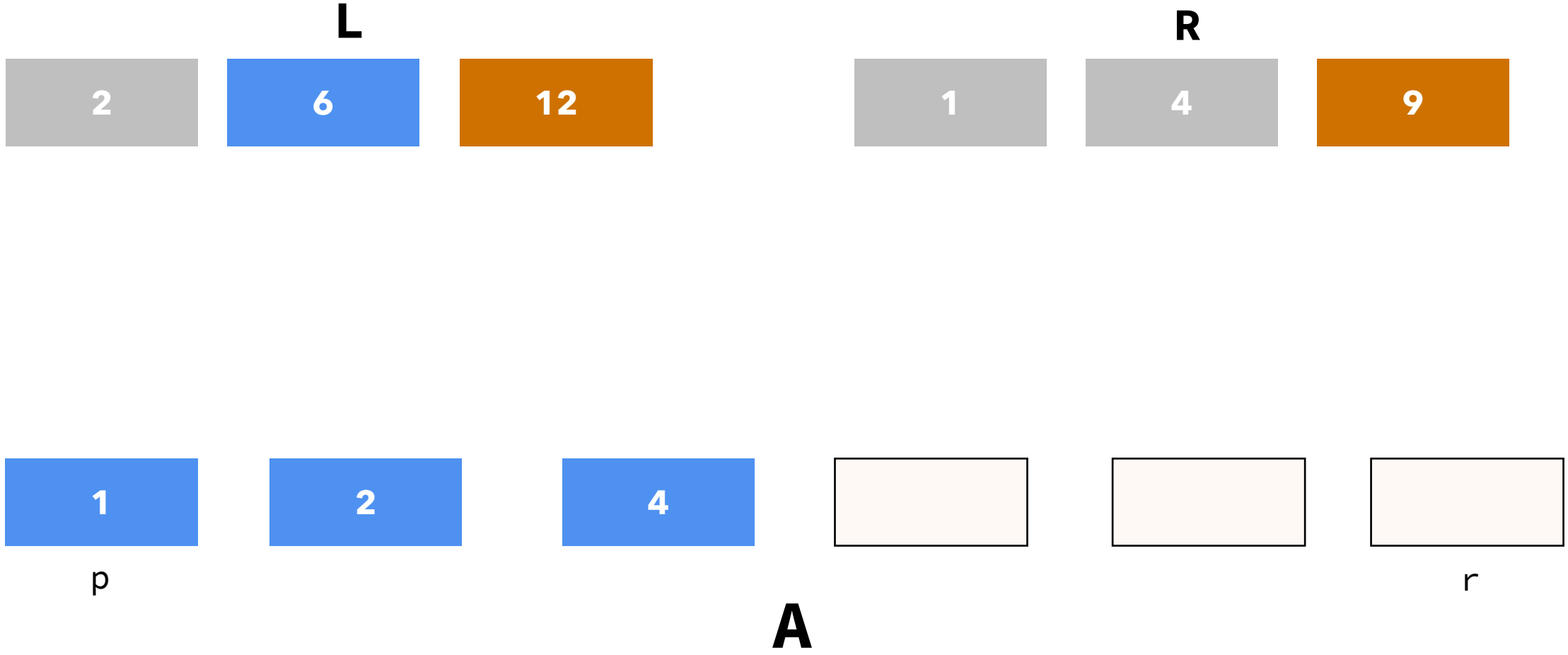
La procedura *merge*



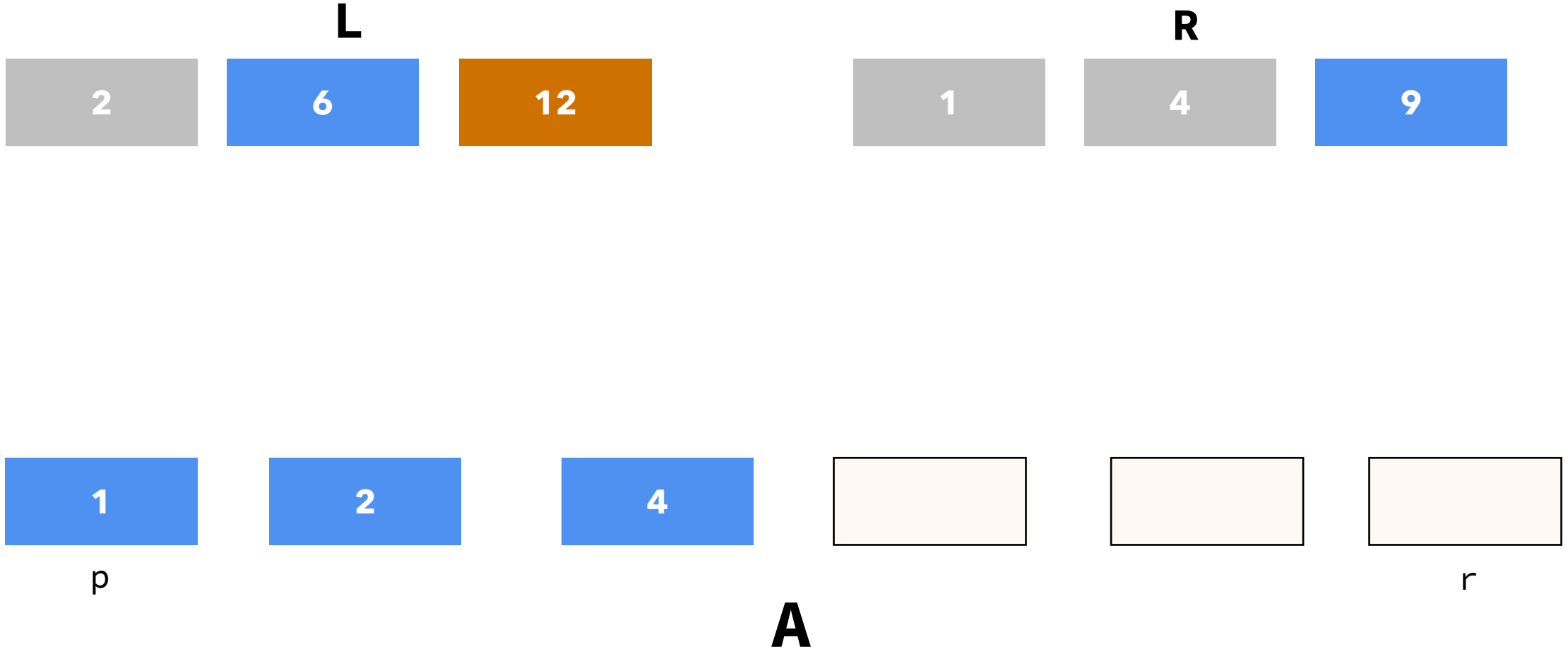
La procedura *merge*



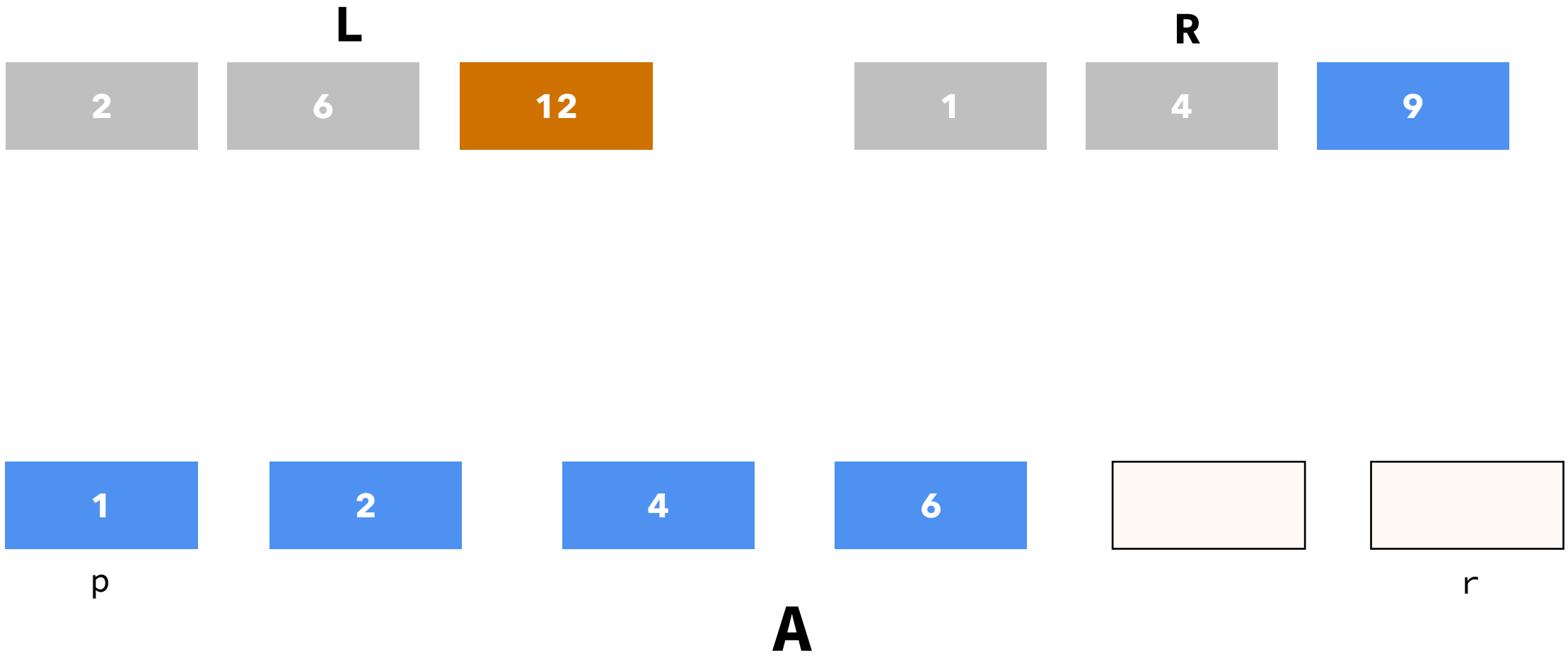
La procedura *merge*



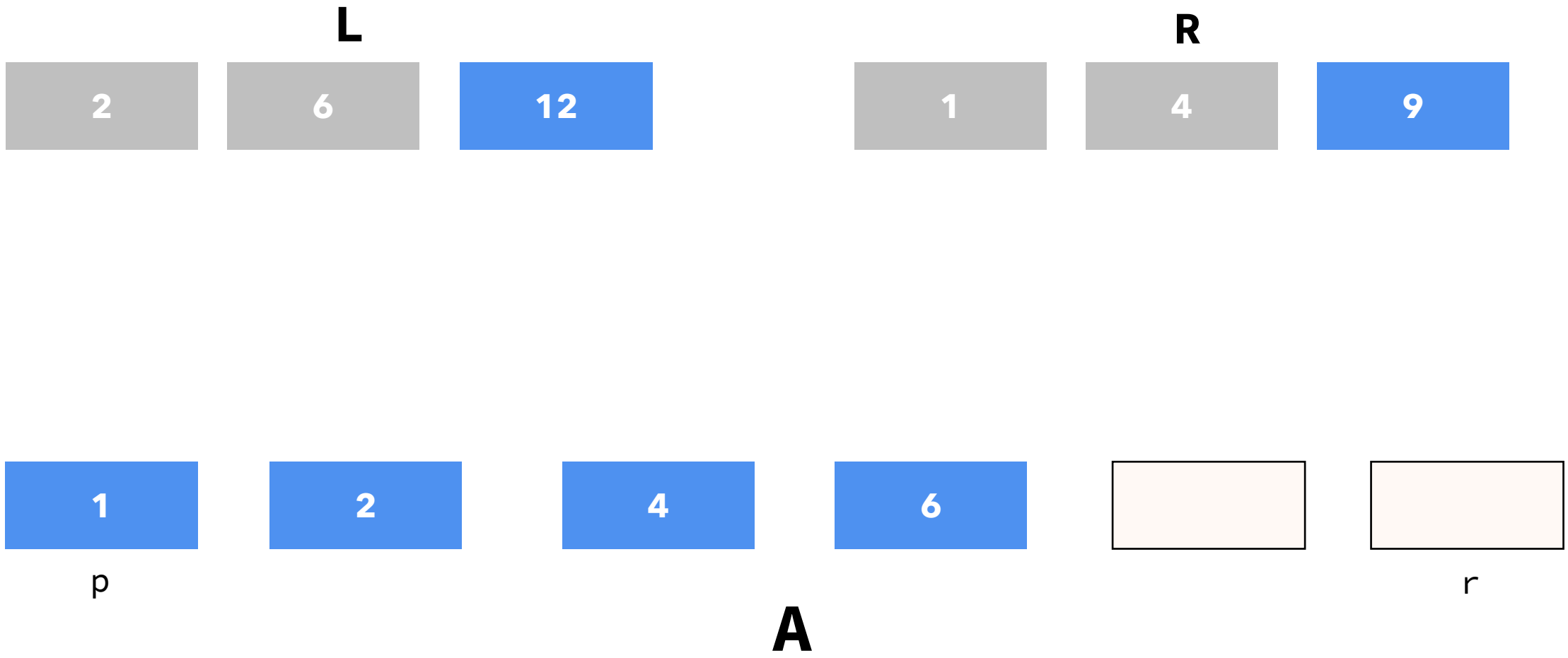
La procedura *merge*



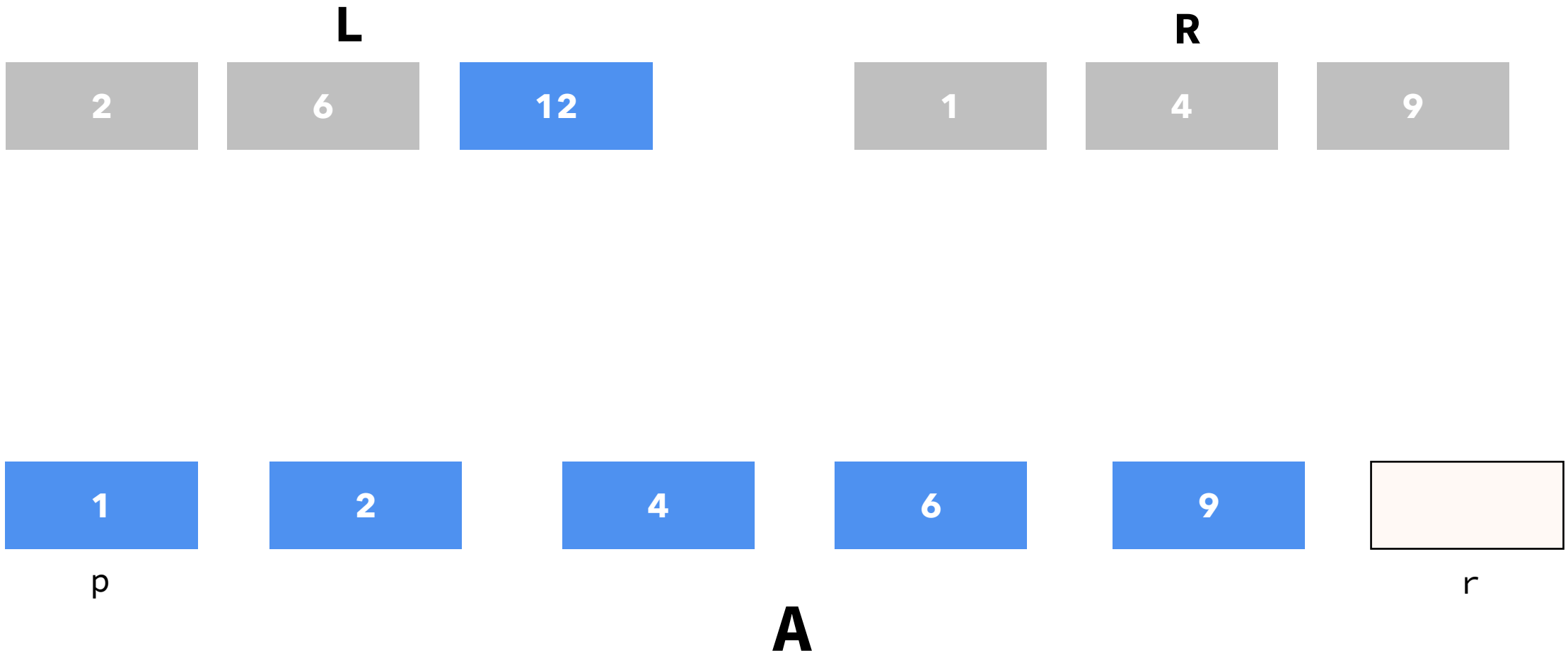
La procedura *merge*



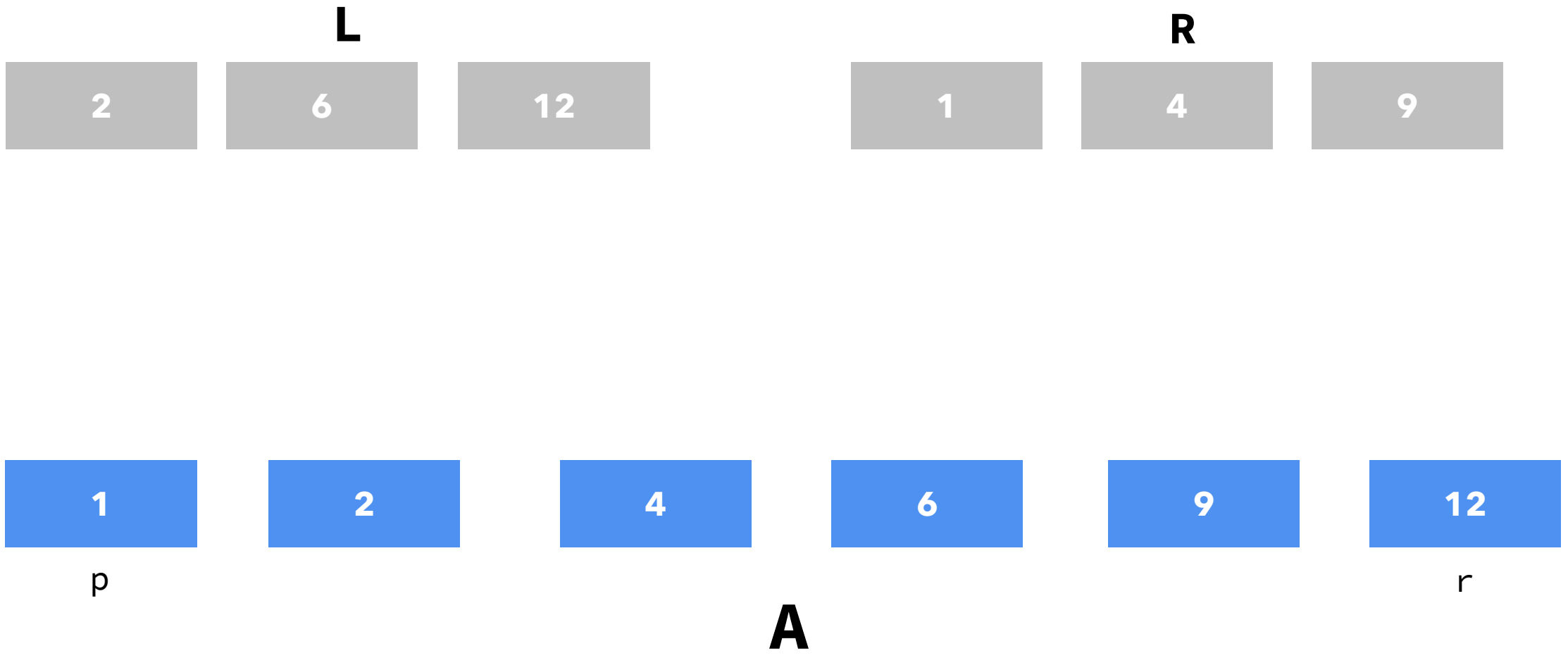
La procedura *merge*



La procedura *merge*



La procedura *merge*



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 nella pila vuota a destra

La procedura *merge*

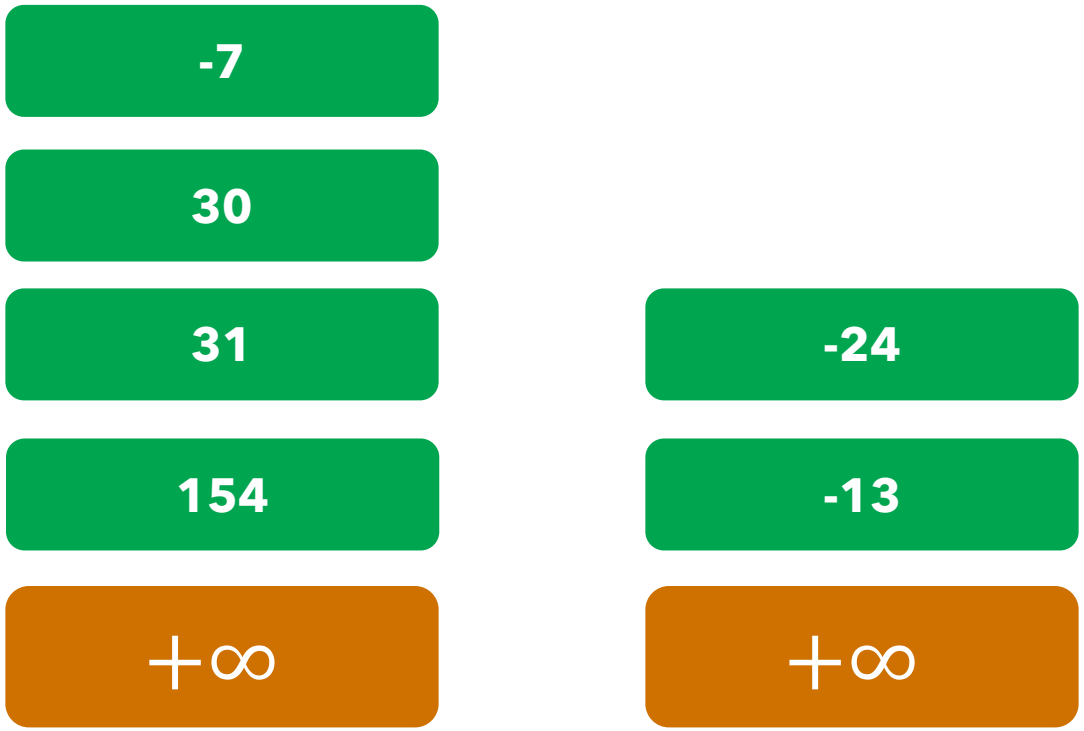
- Per scrivere la procedura *merge*, bisogna prestare molta attenzione a questo fatto: procedendo con la fusione dei due array, ad un certo punto ci si ritrova sempre con uno dei due array vuoto, ossia completamente *sistemato*
- Consideriamo un esempio in cui questo fatto è evidente. Mergiamo due mazzi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

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 nella pila vuota a destra

La procedura *merge*

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



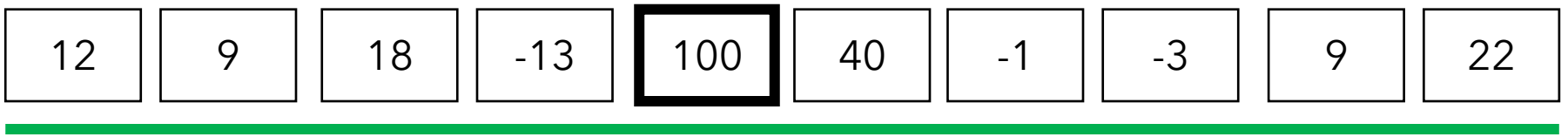
La procedura *merge*

```
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

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

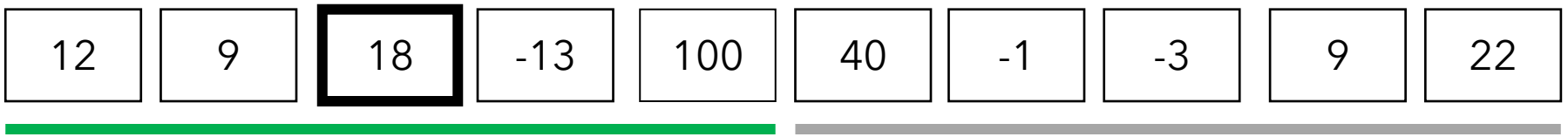
Merge sort



applichiamo Merge sort sulla fetta verde

l'elemento medio della fetta è evidenziato,
le fette grigie sono in sospeso

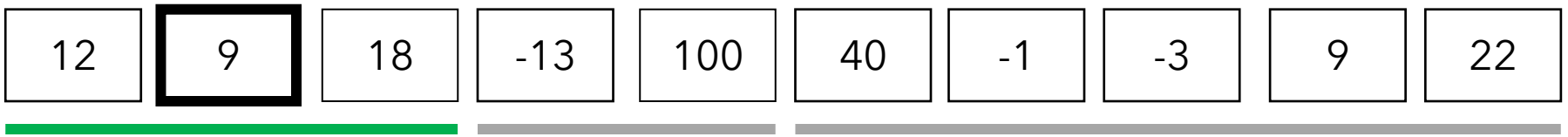
Merge sort



applichiamo Merge sort sulla fetta verde

l'elemento medio della fetta è evidenziato,
le fette grigie sono in sospeso

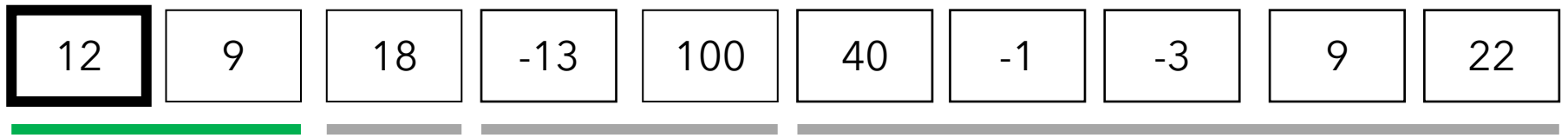
Merge sort



applichiamo Merge sort sulla fetta verde

l'elemento medio della fetta è evidenziato,
le fette grigie sono in sospeso

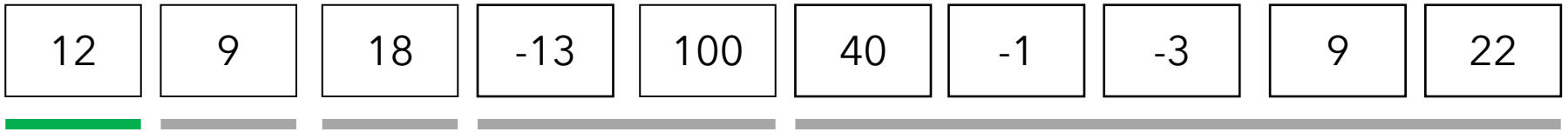
Merge sort



applichiamo Merge sort sulla fetta verde

l'elemento medio della fetta è evidenziato

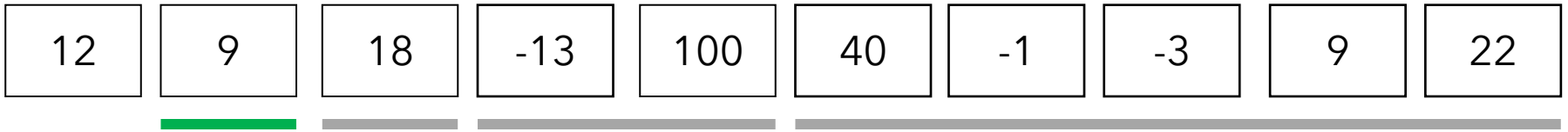
Merge sort



applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

Merge sort

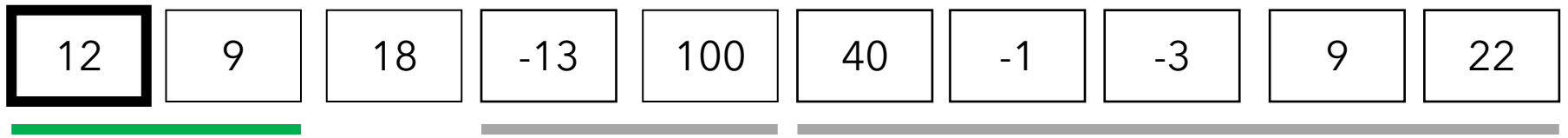


applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

l'elemento medio della fetta è evidenziato

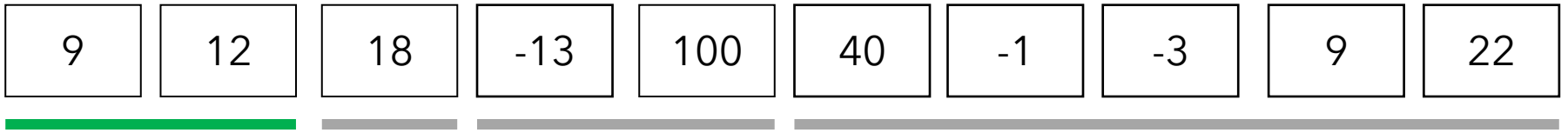
Merge sort



applichiamo merge sulla fetta verde

l'elemento medio della fetta è evidenziato

Merge sort



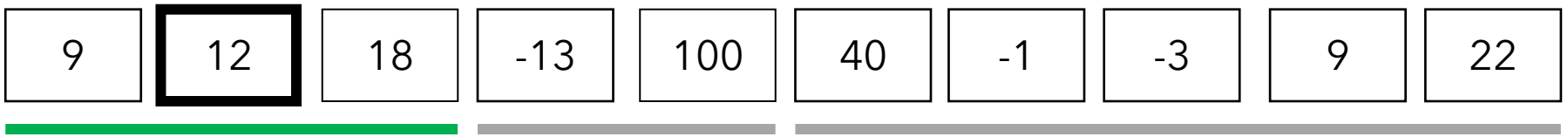
la fetta verde è ordinata

Merge sort

applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

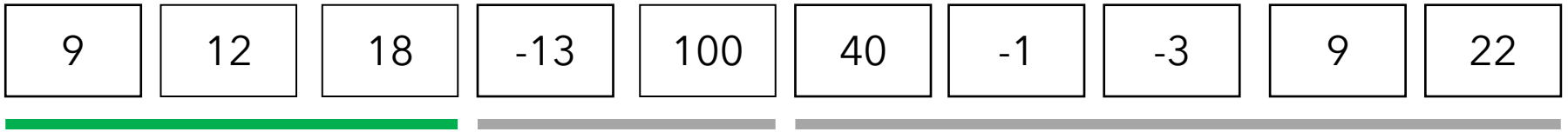
Merge sort



applichiamo merge sulla fetta verde

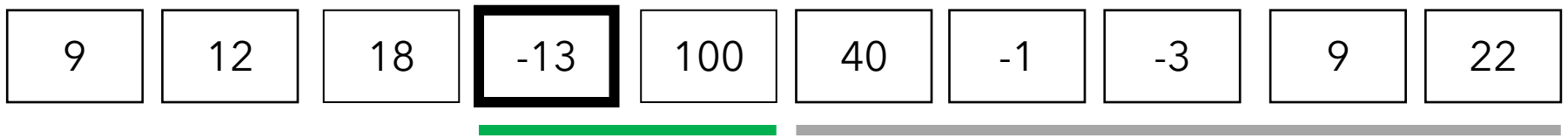
l'elemento medio della fetta è evidenziato

Merge sort



la fetta verde è ordinata

Merge sort



applichiamo Merge sort sulla fetta verde

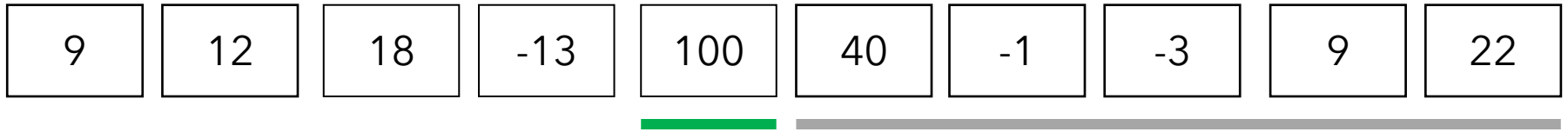
l'elemento medio della fetta è evidenziato

Merge sort

applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata niente da fare

Merge sort

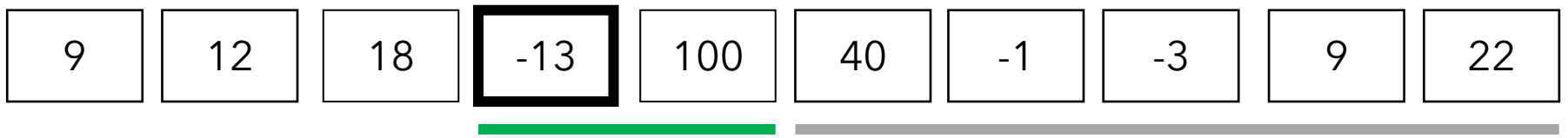


applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

l'elemento medio della fetta è evidenziato

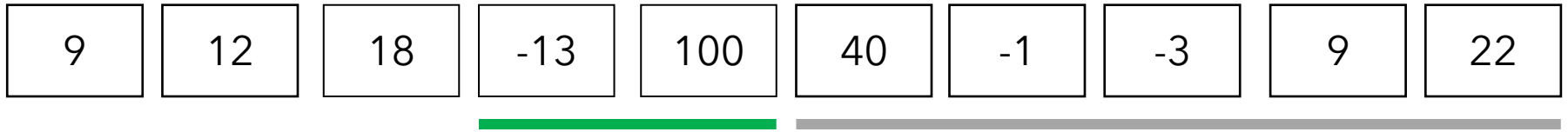
Merge sort



applichiamo merge sulla fetta verde

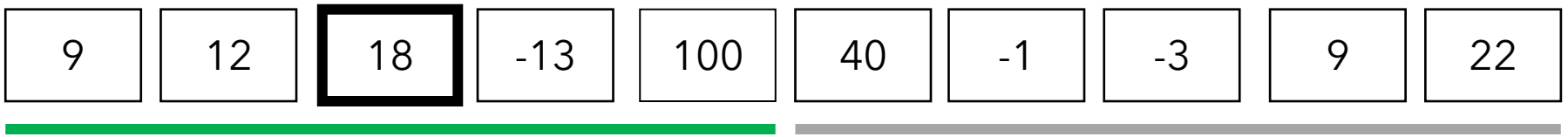
l'elemento medio della fetta è evidenziato

Merge sort



la fetta verde è ordinata

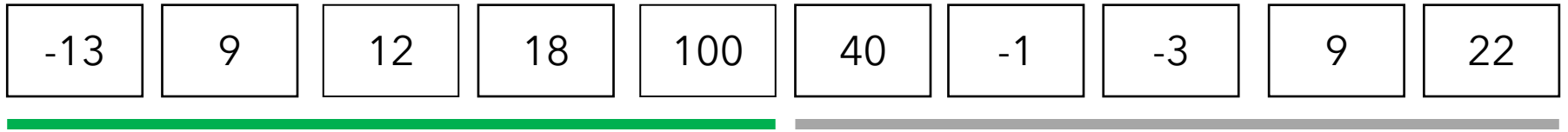
Merge sort



applichiamo merge sulla fetta verde

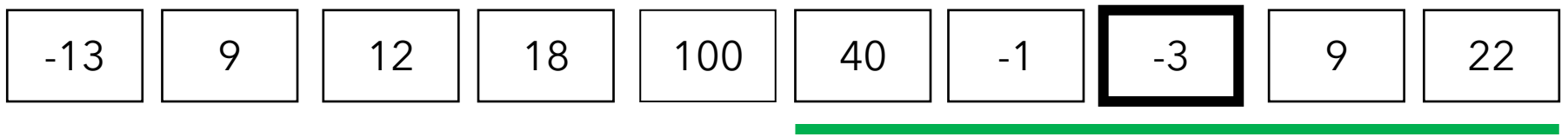
l'elemento medio della fetta è evidenziato

Merge sort



la fetta verde è ordinata

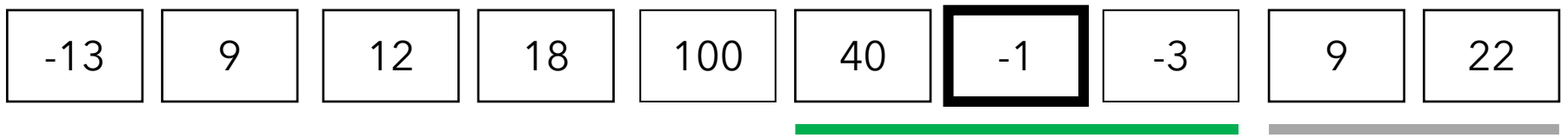
Merge sort



applichiamo Merge sort sulla fetta verde

l'elemento medio della fetta è evidenziato

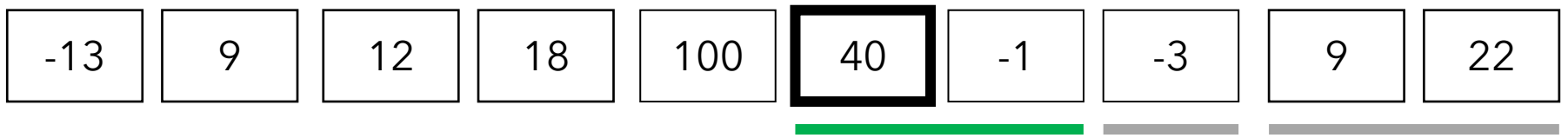
Merge sort



applichiamo Merge sort sulla fetta verde

l'elemento medio della fetta è evidenziato

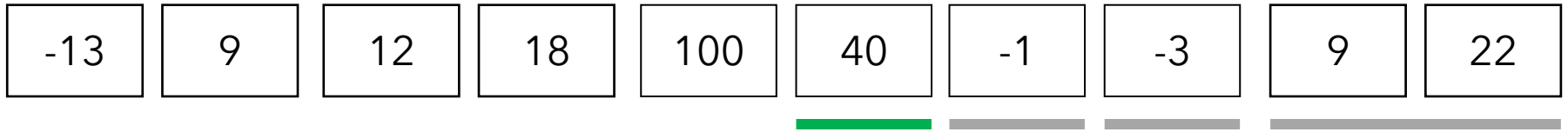
Merge sort



applichiamo Merge sort sulla fetta verde

l'elemento medio della fetta è evidenziato

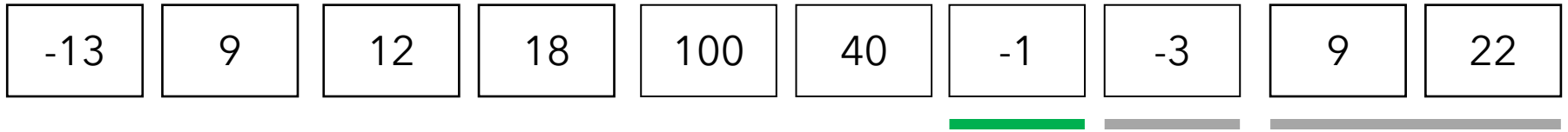
Merge sort



applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

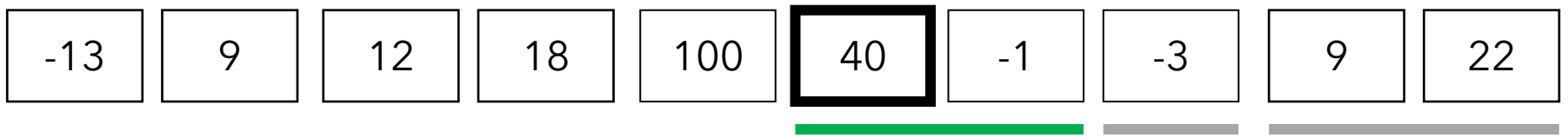
Merge sort



applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

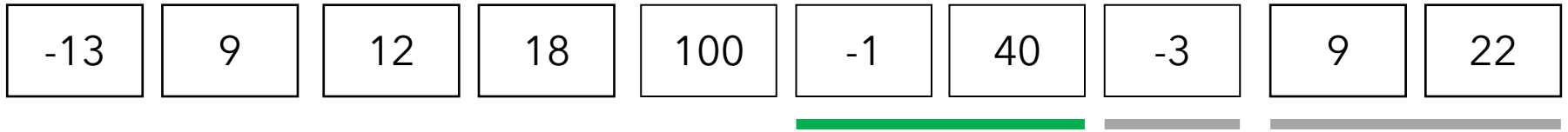
Merge sort



applichiamo merge sulla fetta verde

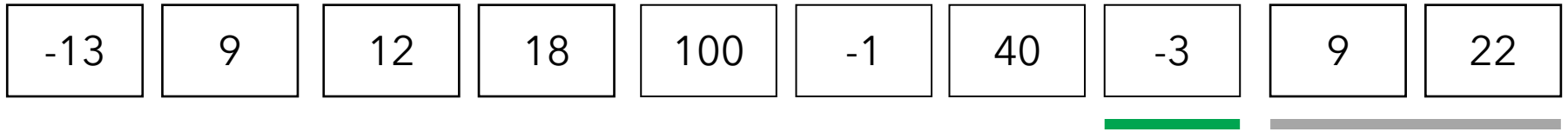
l'elemento medio della fetta è evidenziato

Merge sort



la fetta verde è ordinata

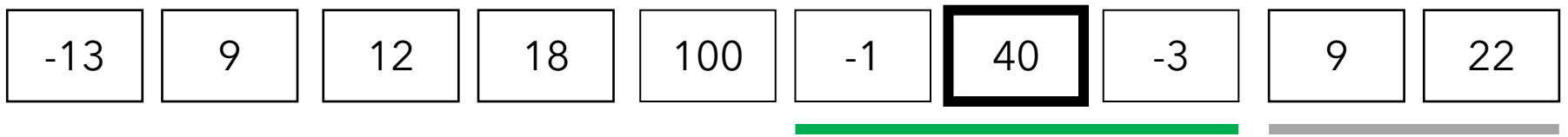
Merge sort



applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

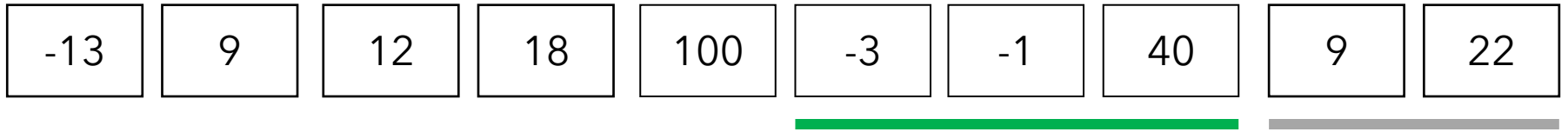
Merge sort



applichiamo merge sulla fetta verde

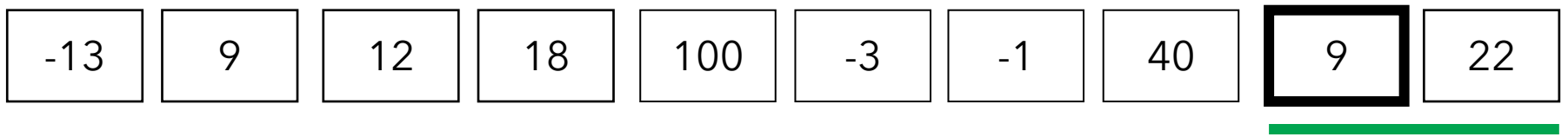
l'elemento medio della fetta è evidenziato

Merge sort



la fetta verde è ordinata

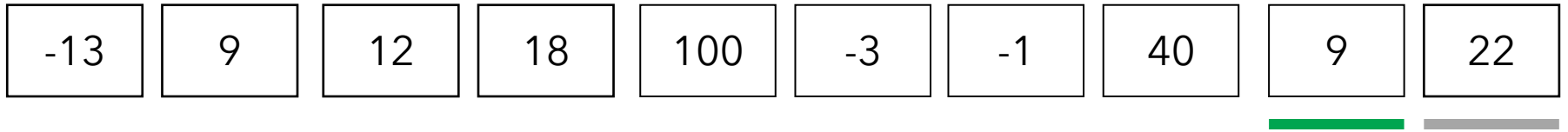
Merge sort



applichiamo Merge sort sulla fetta verde

l'elemento medio della fetta è evidenziato,
le fette grigie sono in sospeso

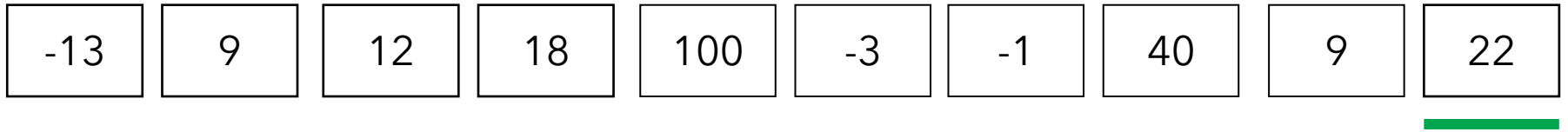
Merge sort



applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

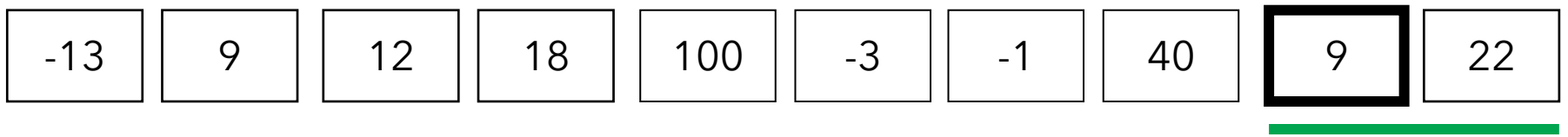
Merge sort



applichiamo Merge sort sulla fetta verde

fetta di 1 elemento: già ordinata, niente da fare

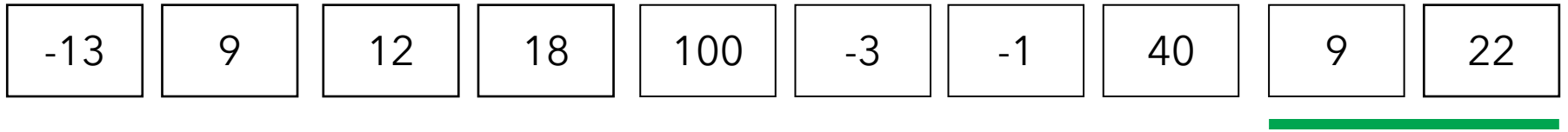
Merge sort



applichiamo merge sulla fetta verde

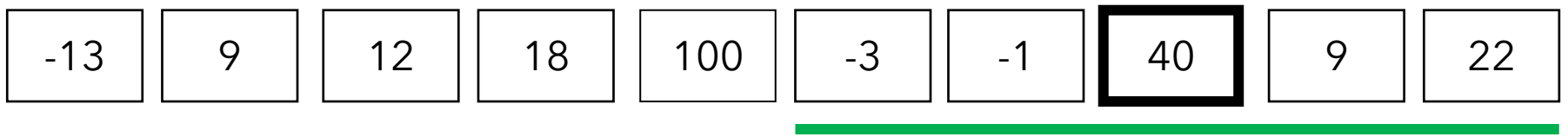
l'elemento medio della fetta è evidenziato

Merge sort



la fetta verde è ordinata

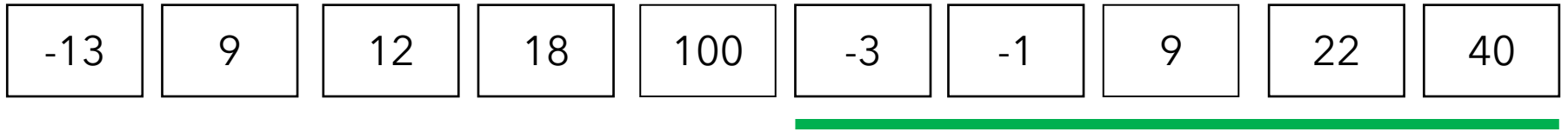
Merge sort



applichiamo Merge sort sulla fetta verde

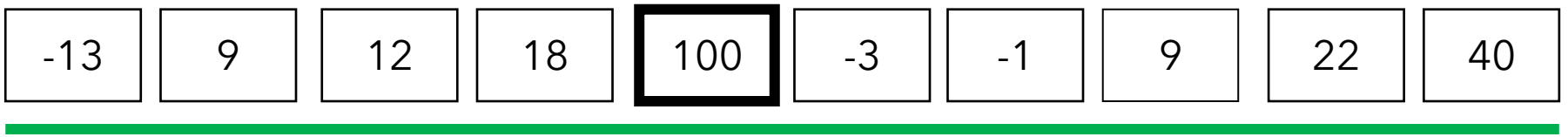
l'elemento medio della fetta è evidenziato

Merge sort



la fetta verde è ordinata

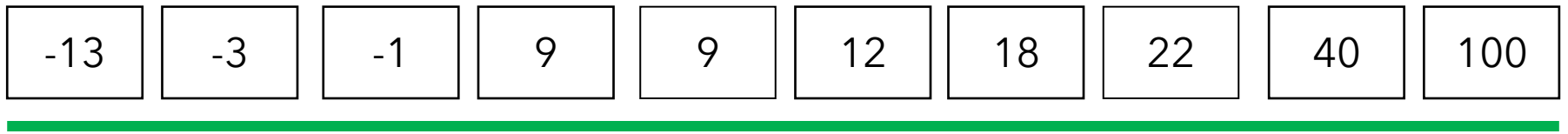
Merge sort



applichiamo merge sulla fetta verde

l'elemento medio della fetta è evidenziato

Merge sort



la fetta verde è ordinata

l'array iniziale è ordinato

Merge sort

```
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)
```

Da vedere a casa

- **Merge Sort vs Quick Sort**