Code review e note sulle *slice* di Go

Violetta Lonati

Università degli studi di Milano
Dipartimento di Informatica

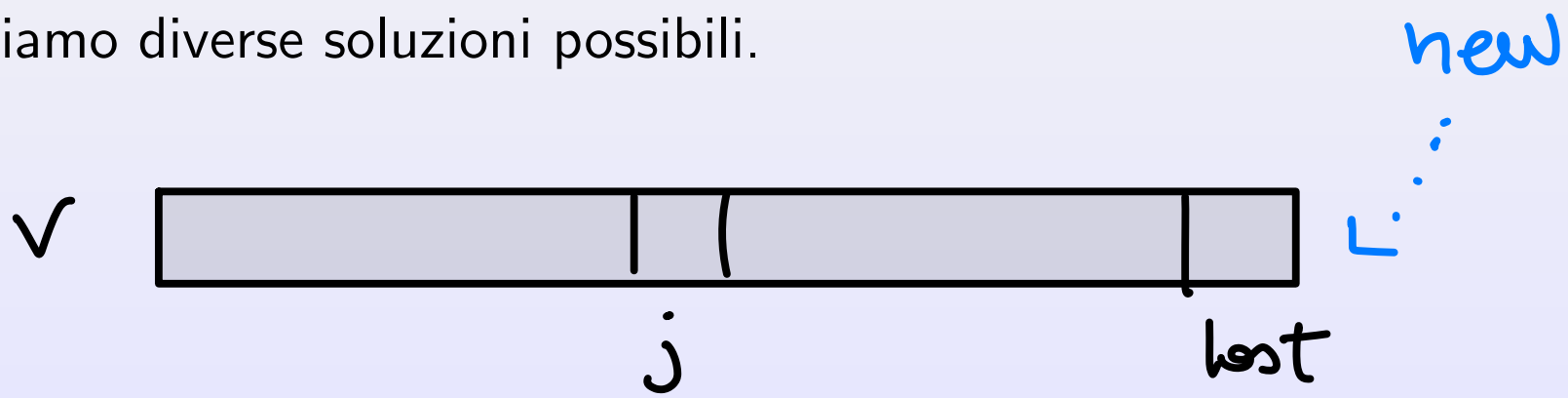
Laboratorio di algoritmi e strutture dati
Corso di laurea in Informatica

InsertionSort

Abbiamo una slice `v` ordinata, e vogliamo inserire un nuovo elemento `new`. Una volta trovata la posizione `j` in cui inserirlo, bisogna fare lo *shift* verso destra di tutti gli elementi che si trovano a destra della posizione `j` (inclusa).

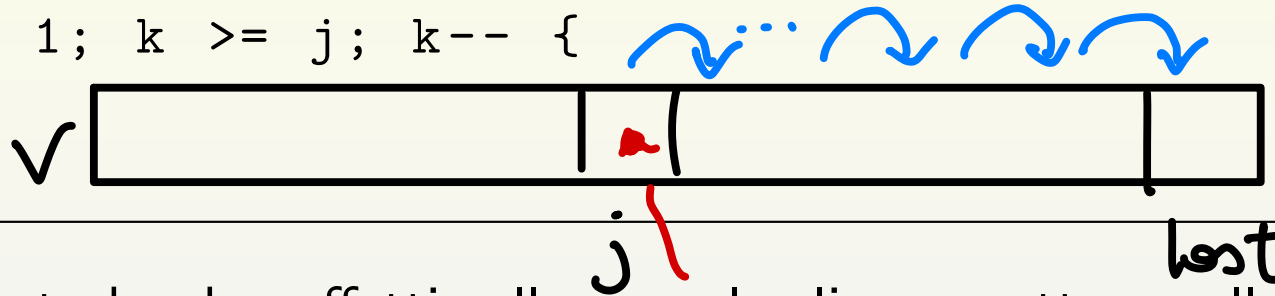
Sia `last` l'ultimo indice di `v` prima dell'inserimento di `new`. Dunque dobbiamo spostare gli elementi che si trovano nelle posizioni dalla `j` alla `last` comprese.

Vediamo diverse soluzioni possibili.



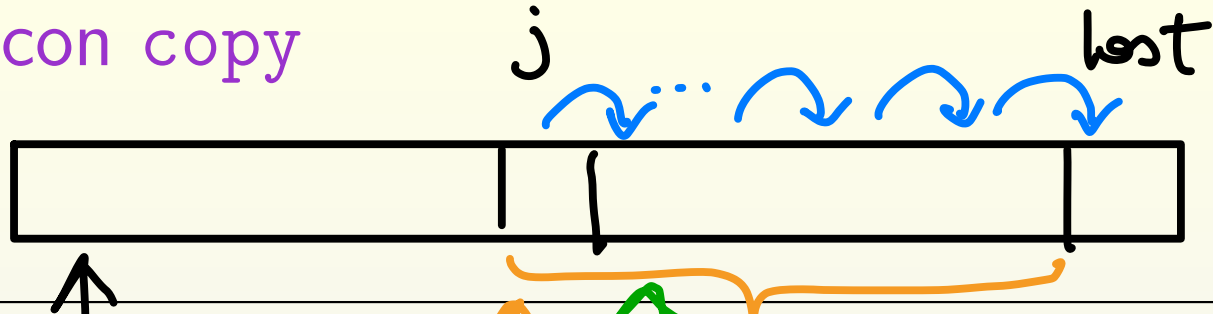
InsertionSort: copia passo-passo

```
v = append(v, v[last])
for k := last - 1; k >= j; k-- {
   v[k+1] = v[k]
}
```



- Il primo assegnamento ha due effetti: allungare la slice e mettere nella posizione `last+1` l'elemento `v[last]`
- gli elementi da `j` a `last-1` vengono copiati nelle posizioni da `j+1` a `last` (non c'è bisogno di spostare l'elemento nella posizione `last`!)
- gli elementi vengono spostati a partire da quello più a sinistra, per evitare di sovrascriverli
- `v[j]` non è cambiato, e alla fine andrà sostituito con `new`
- il costo è proporzionale al numero di elementi da spostare, cioè `last-j`

InsertionSort: copia con copy

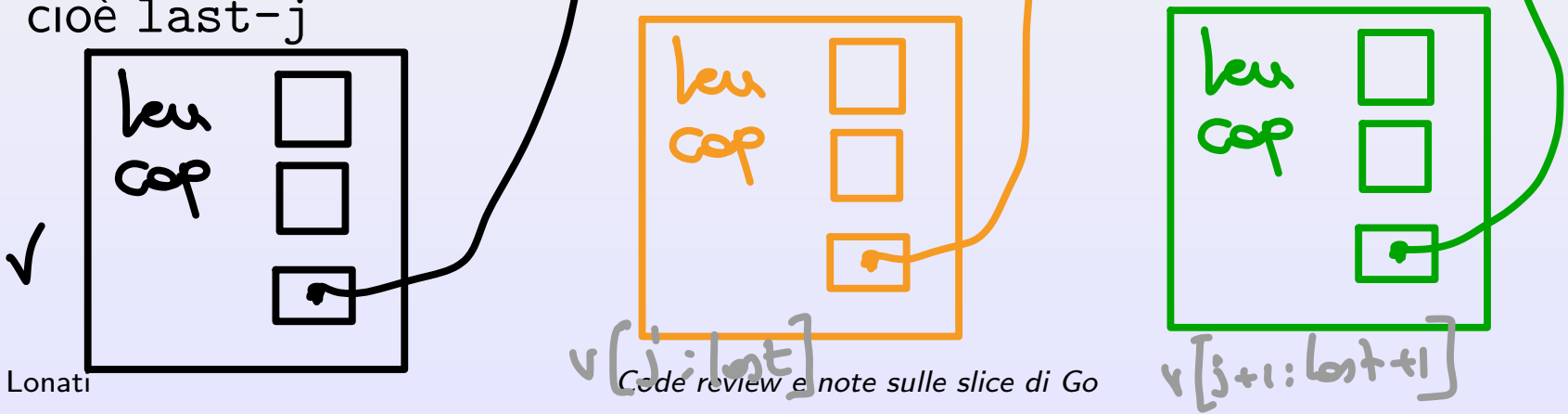



```
for k := last - 1; k >= j; k-- {
   v[k+1] = v[k]
}
```

si può abbreviare con:

```
copy(v[j+1:last+1], v[j:last])
```

- le operazioni svolte sono meno visibili, ma non sono di meno!
- il costo è sempre proporzionale al numero di elementi da spostare, cioè `last-j`



InsertionSort: concatenazione di sottoslice

Sfruttando la funzione `append` possiamo concatenare tre parti:

- la parte di slice fino all'indice `j` escluso,
- il nuovo elemento
- la parte rimanente di slice

Usiamo una slice di supporto, inizialmente vuota:

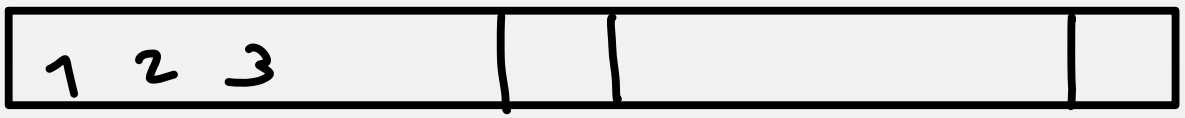
```
res := make([]int, 0)
```

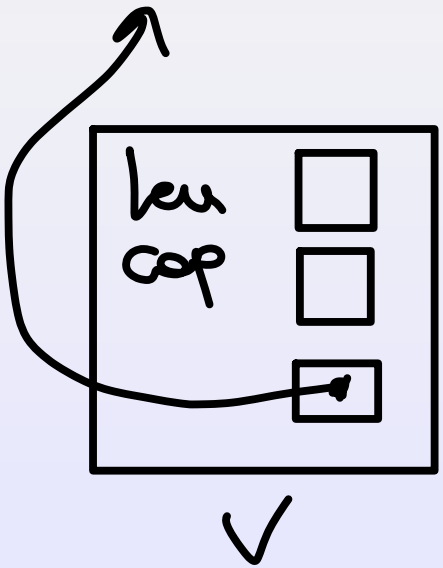
```
res = append(res, v[:j]...)
res = append(res, new)
res = append(res, v[j:]...)
v = res
```

- Qual è il costo di questa soluzione?
- Quali sono le parti più onerose?

InsertionSort: concatenazione di sottoslice

```
res := make([]int, 0)
res = append(res, v[:j]...)
res = append(res, new)
res = append(res, v[j:]...)
v = res
```

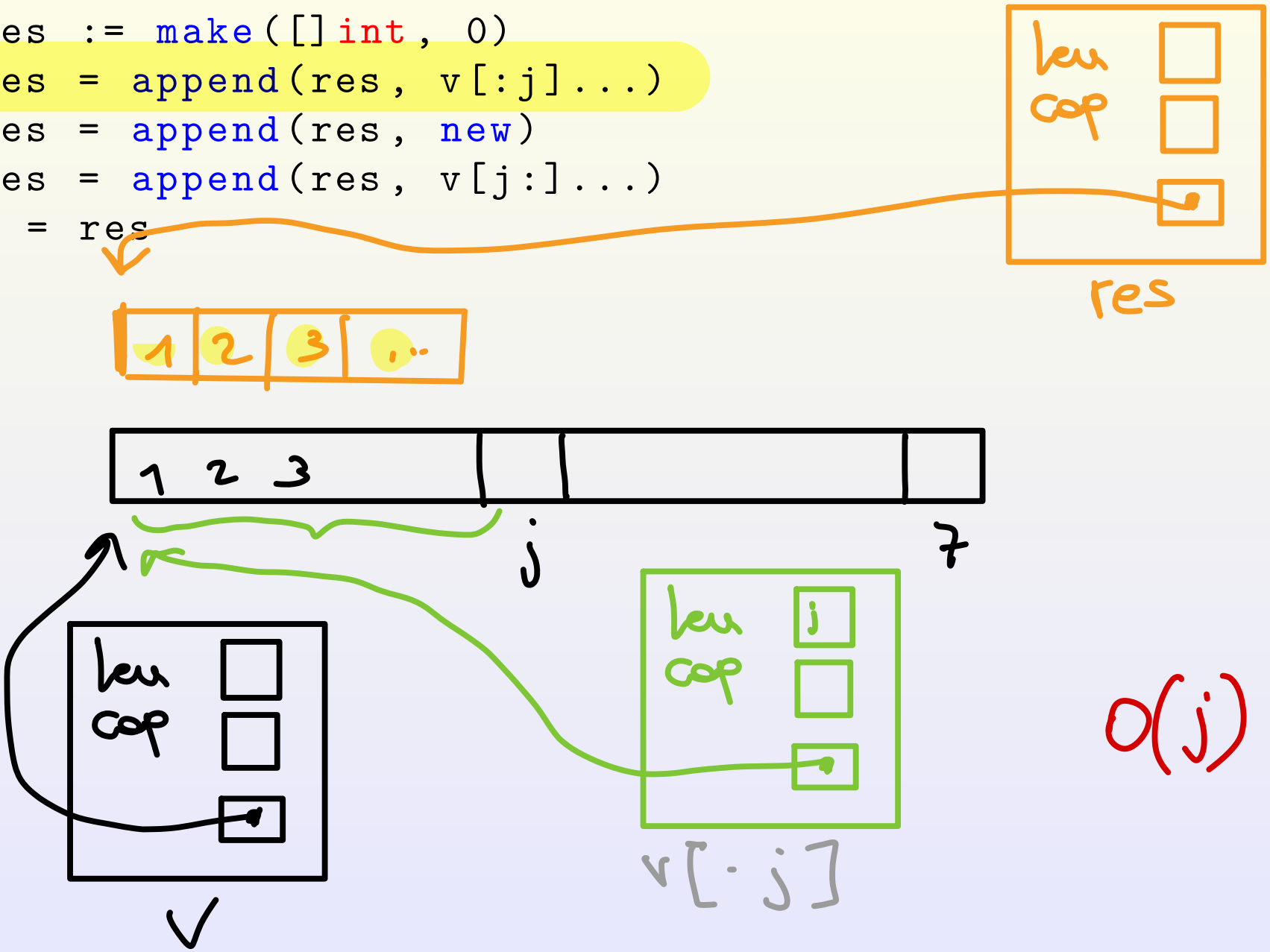




O(1)

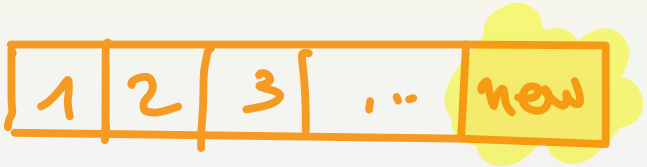
InsertionSort: concatenazione di sottoslice

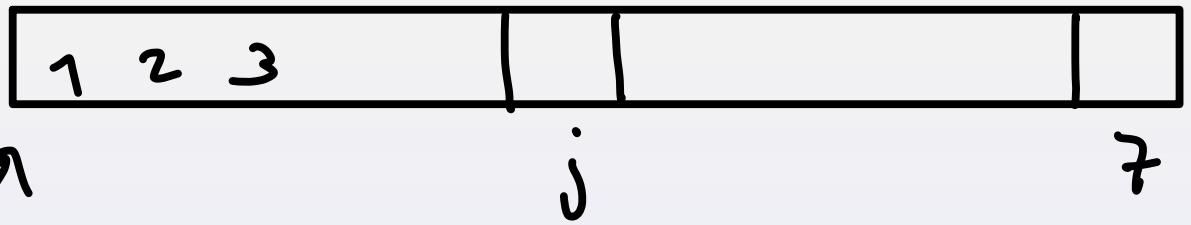
```
res := make([]int, 0)
res = append(res, v[:j]...)
res = append(res, new)
res = append(res, v[j:]...)
v = res
```

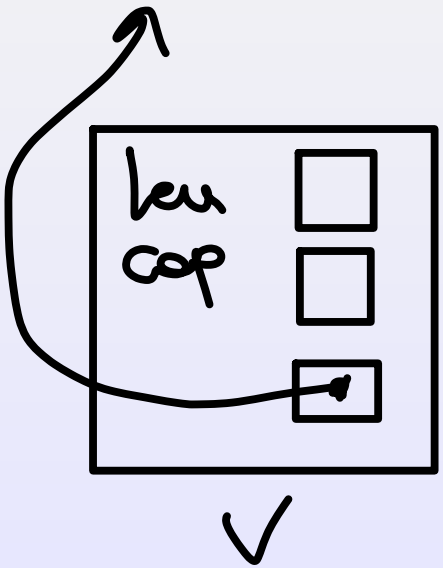


InsertionSort: concatenazione di sottoslice

```
res := make([]int, 0)
res = append(res, v[:j]...)
res = append(res, new)
res = append(res, v[j:]...)
v = res
```



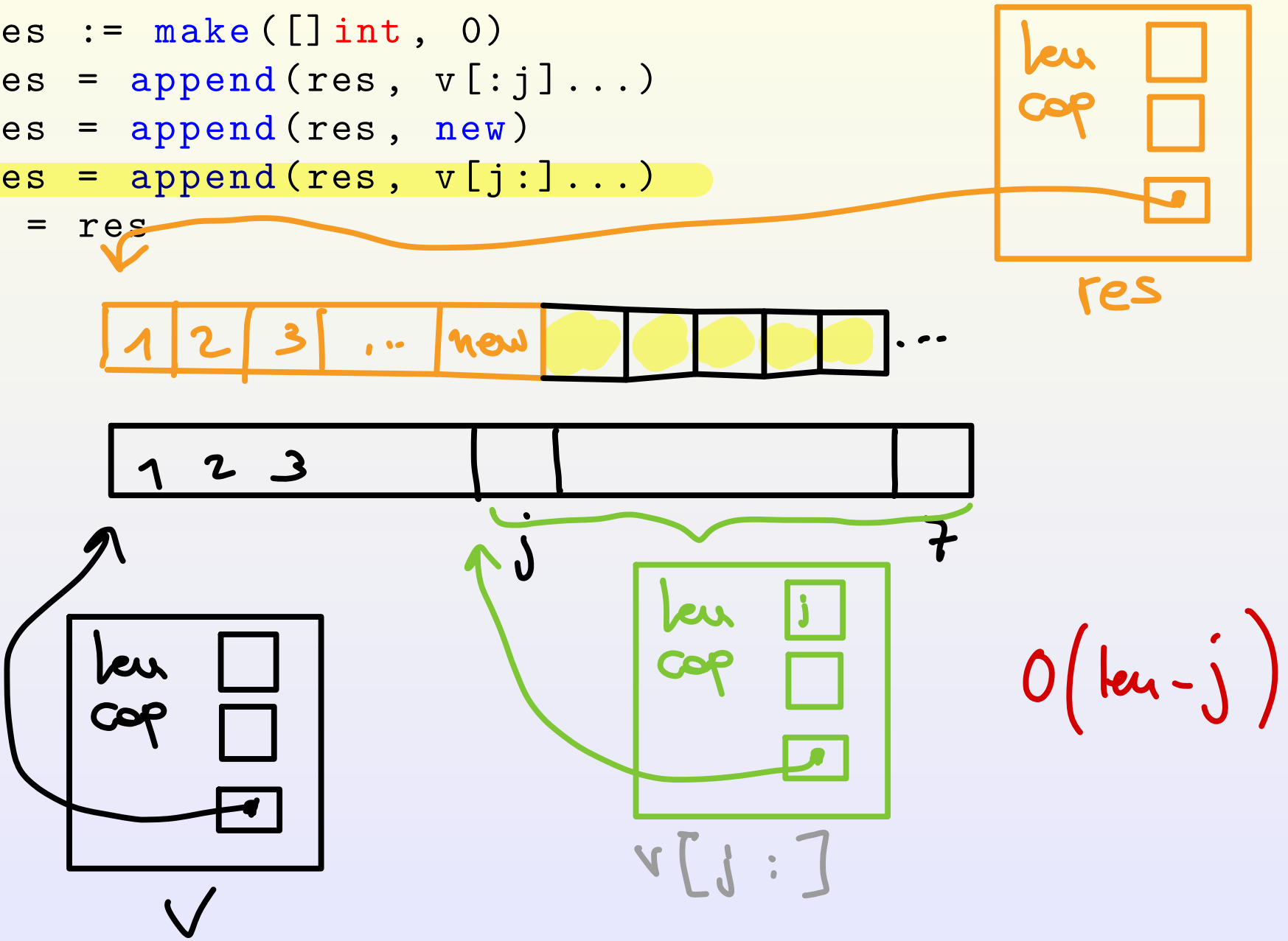




O(1)

InsertionSort: concatenazione di sottoslice

```
res := make([]int, 0)
res = append(res, v[:j]...)
res = append(res, new)
res = append(res, v[j:]...)
v = res
```



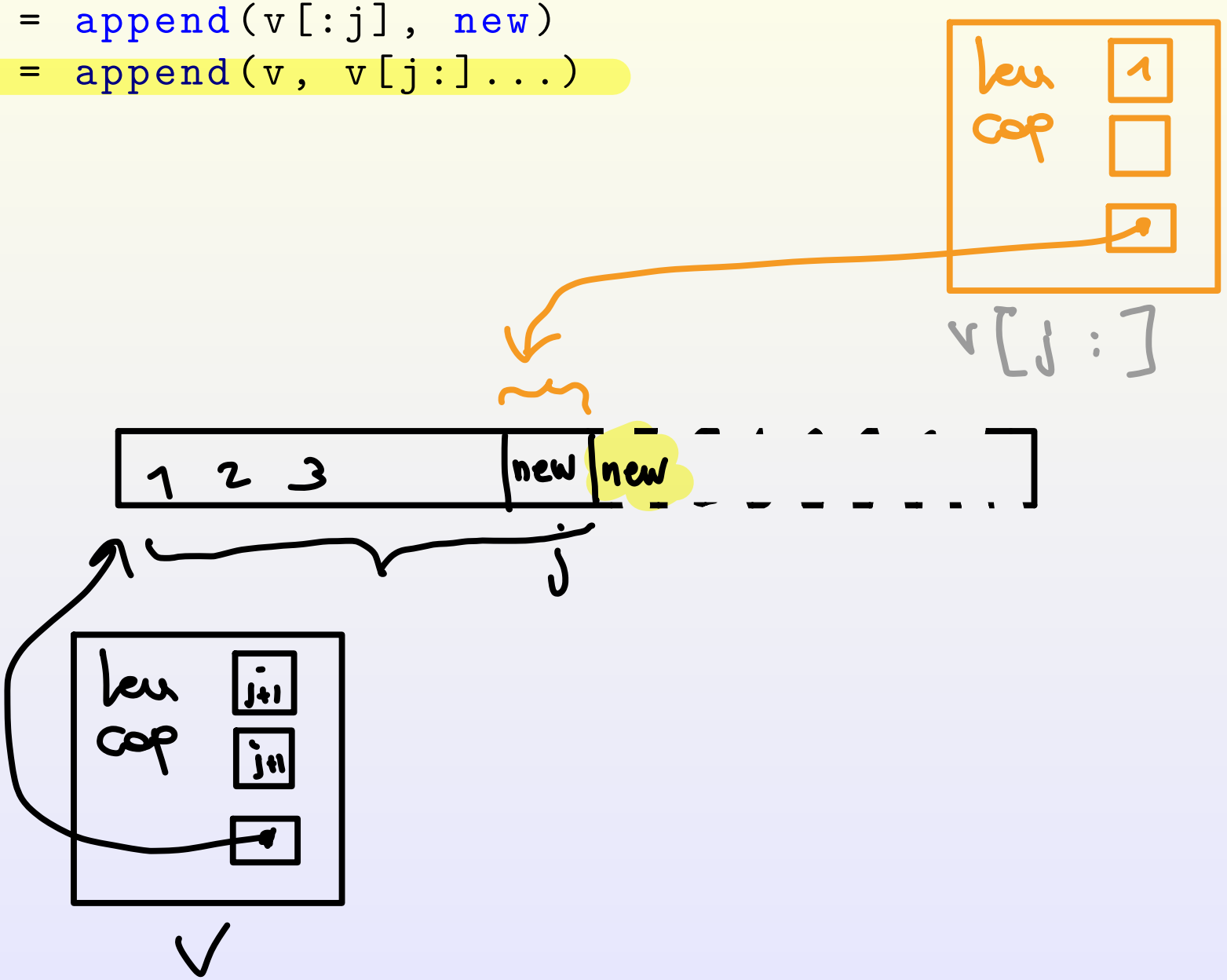
InsertionSort: concatenazione di sottoslice

```
res := make([]int, 0)
res = append(res, v[:j]...)
res = append(res, new)
res = append(res, v[j:]...)
v = res
```

- Sembrano 3 assegnamenti, ma il costo è sempre proporzionale al numero di elementi da spostare, cioè `last-j`
- l'`append` con `...` equivale a un ciclo di `append` (una per ogni elemento della seconda slice)

InsertionSort: concatenazione di sottoslice - Attenzione!

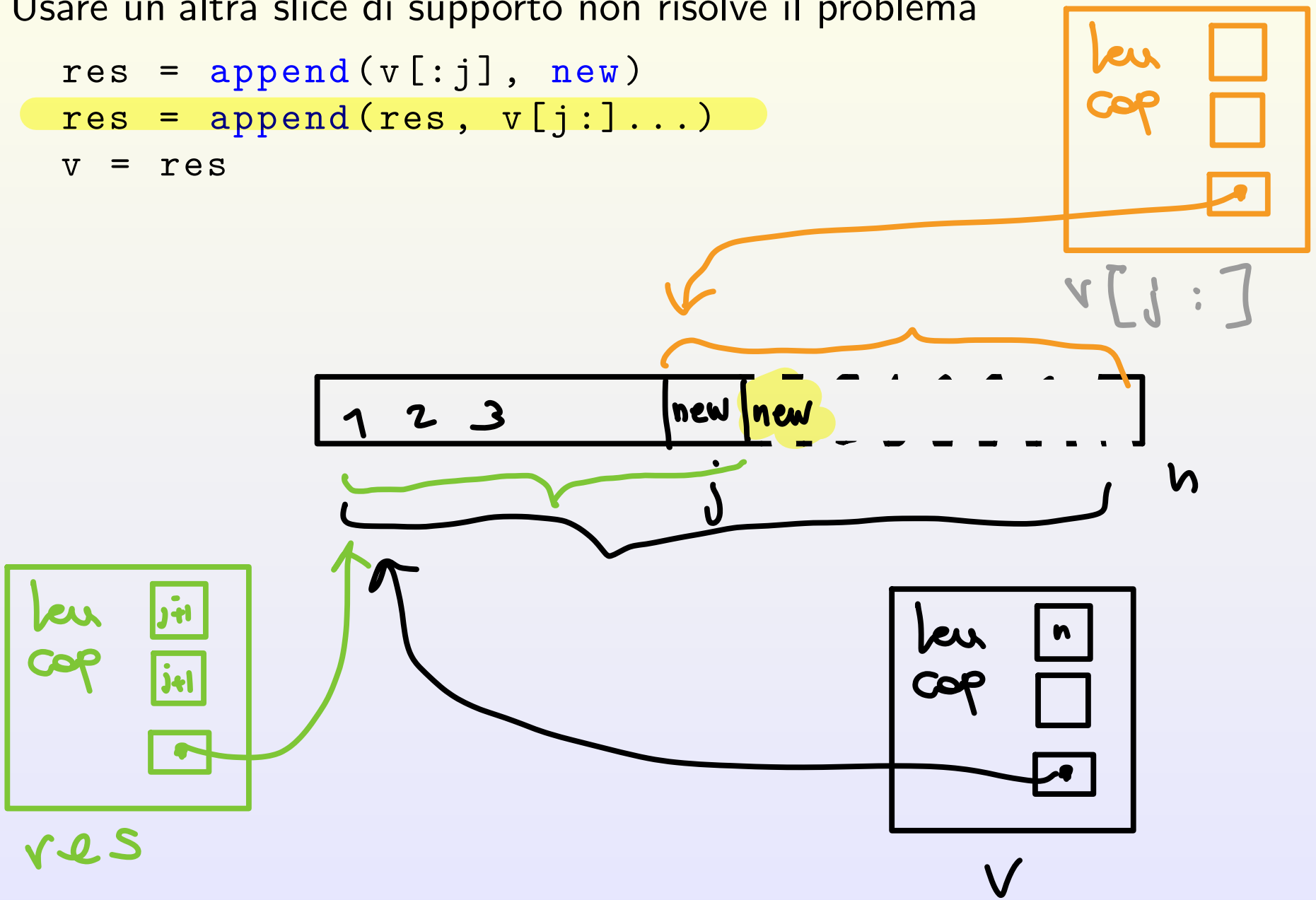
```
v = append(v[:j], new)
v = append(v, v[j:]...)
```



InsertionSort: concatenazione di sottoslice - Attenzione!

Usare un'altra slice di supporto non risolve il problema

```
res = append(v[:j], new)
res = append(res, v[j:]...)
v = res
```



MergeSort

```
func mergeSort(v []int) []int {
  if len(v) == 1 {
    return v
  }
  n := len(v) / 2

  mergeSort(v[:n])
  mergeSort(v[n:])
  copy(v, merge(v[:n], v[n:]))
  return v
}
```

La funzione `merge(a, b []int) (res []int)` costruisce e restituisce una slice contenente tutti gli elementi di `a` e `b` in ordine.
Si compone di due parti: finita la parte di "fusione" vera e propria, bisogna copiare la "coda" della slice rimasta.

Merge - passo passo

Alloco una slice della lunghezza necessaria e inserisco gli elementi uno alla volta.

```
res = make([]int, len(a)+len(b))
i, j, k := 0, 0, 0
for i < len(a) && j < len(b) {
  if a[i] < b[j] {
    res[k] = a[i]
    i++
  } else {
    res[k] = b[j]
    j++
  }
  k++
}
```

- Costo in tempo? Spazio occupato?

Coda - passo passo

```
if i < len(a) {
    for _, el := range a[i:] {
        res[k] = el
        k++
    }
}

if j < len(b) {
    for _, el := range b[j:] {
        res[k] = el
        k++
    }
}
```

MergeSort con append

```
func merge(a, b []int) (res []int) {
  i, j := 0, 0
  for i < len(a) && j < len(b) {
    if a[i] < b[j] {
      res = append(res, a[i])
      i++
    } else {
      res = append(res, b[j])
      j++
    }
  }
  res = append(res, a[i:]...)
  res = append(res, b[j:]...)
}
```

- una coda o due?
- Soluzione più idiomatica
- Costo in tempo? Spazio occupato?

Con append + pre-allocazione

Alloco una slice con la capacità necessaria e inserisco gli elementi uno alla volta.

```
func merge(a, b []int) (res []int) {
  res = make([]int, 0, len(a)+len(b))

  ...
}
```

- La soluzione resta idiomatica
- Costo in tempo? Spazio occupato?

MergeSort

```
func mergeSort(v []int) []int {
    if len(v) == 1 {
        return v
    }
    n := len(v) / 2

    mergeSort(v[:n])
    mergeSort(v[n:])
    copy(v, merge(v[:n], v[n:]))
    return v
}
```

- La funzione `merge(a, b []int) (res []int)` costruisce e restituisce una slice contenente tutti gli elementi di `a` e `b` in ordine.
- A cosa serve la `copy`? È necessaria? Non basterebbe un assegnamento: `v = merge(v[:n], v[n:])` ?