Lab08 – Array/Slice – II parte

Quando si deve rappresentare una sequenza di elementi dello stesso tipo la cui dimensione può variare, è consigliabile utilizzare una slice e lavorare su di essa con la funzione append.

La funzione append è molto versatile e può essere utilizzata per effettuare svariate manipolazioni[1]:

1) Append a slice b to an existing slice a:	`a = append(a, b...)`
2) Copy a slice a to a new slice b:	`b = make([]T, len(a))` `copy(b, a)`
3) Delete item at index i:	`a = append(a[:i], a[i+1:]...)`
4) Cut from index i till j out of slice a:	`a = append(a[:i], a[j:]...)`
5) Extend slice a with a new slice of length j:	`a = append(a, make([]T, j)...)`
6) Insert item x at index i:	`a = append(a[:i], append([]T{x},` `a[i:]...)...)`
7) Insert a new slice of length j at index i:	`a = append(a[:i], append(make([]T,` `j), a[i:]...)...)`
8) Insert an existing slice b at index i:	`a = append(a[:i], append(b,` `a[i:]...)...)`
9) Pop highest element from stack:	`x, a = a[len(a)-1], a[:len(a)-1]`
10) Push an element x on a stack:	`a = append(a, x)`

Esercizio 1 – Analisi di codice

```go
func main() {

  const DIMENSIONE = 10

  var a []int

  for i := 0; i < DIMENSIONE; i++ {
     a = append(a, i)
     fmt.Println("Iterazione", i, ":", a, len(a), cap(a))
```

[1] The Way To Go: A Thorough Introduction To The Go Programming Language
iUniverse, Incorporated ©2012
ISBN:1469769166 9781469769165

```
    }


    fmt.Println(strings.Repeat("=", 10))
    fmt.Println(a, len(a), cap(a))
    a = a[:cap(a)]
    fmt.Println(a, len(a), cap(a))
    a = a[:DIMENSIONE]

    b := append(a[DIMENSIONE/2:], a[:DIMENSIONE/2]...)

    fmt.Println(strings.Repeat("=", 10))
    fmt.Println(a, len(a), cap(a), b, len(b), cap(b))
    a = a[:cap(a)]
    fmt.Println(a, len(a), cap(a), b, len(b), cap(b))
    a = a[:DIMENSIONE]


    c := append(a[:DIMENSIONE/4], a[1 + 3*DIMENSIONE/4:]...)

    fmt.Println(strings.Repeat("=", 10))
    fmt.Println(a, len(a), cap(a), b, len(b), cap(b), c, len(c), cap(c))
}
```

Esercizio 2 – Array di numeri random

Creare un programma che:

1) Riceva come argomento da riga di comando un numero intero chiamato *soglia*
2) Generi una serie di numeri interi casuali compresi tra 0 e 100 fermandosi al primo numero inferiore di *soglia*
3) Salvi tutti i numeri generati in una slice
4) Stampi tutti i valori generati
5) Stampi solamente i valori della slice che sono superiori a *soglia*

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run random.go 20
Valori generati [21 72 44 64 30 13]
Valori sopra soglia: [21 72 44 64 30]
```

SUGGERIMENTO

Dato che non è nota a priori la dimensione della slice, utilizzate la funzione `append(slice, elemento)` per modificare in modo dinamico la slice.

Esercizio 3 – Utilizzo delle funzioni copy e append – Minimo/Massimo/Media relativi a sequenze di numeri di lunghezza ignota

Scrivere un programma che chieda all'utente di inserire una sequenza di numeri (uno per riga). La fase di inserimento dei numeri termina alla lettura di una riga vuota.

Una volta terminata la fase di inserimento, si supponga siano stati inseriti dall'utente $n >= 3$ numeri che definiscono la sequenza $S_0 = seq_1, seq_2, ..., seq_{n-1}, seq_n$.

Il programma definisce a partire da S_0 le seguenti sequenze:

$S_1 = 2^1seq_2, ..., 2^{n-2}seq_{n-1}, 2^{n-1}seq_n, 2^{-1}seq_2, ..., 2^{-(n-2)}seq_{n-1}, 2^{-(n-1)}seq_n,$

$S_2 = seq_1, seq_2, ..., seq_k, seq_{n-k+1}, ..., seq_{n-1}, seq_n$ (con $k = n/3$)

$S_3 = seq_1, seq_2, ..., seq_{n/3}, a_0, a_1, a_2, ..., a_9, seq_{2(n/3)}, ..., seq_{n-1}, seq_n$

dove $a_0, a_1, a_2, ..., a_9$ sono 10 numeri casuali generati nell'intervallo [-10000; 10000].

Tutte le sequenze S_i (con $i=0,...,3$) devono essere memorizzate in una sequenza di sequenze SS.

Utilizzando una funzione con segnatura

```
func esaminaSequenza(nomeSequenza string, sl []float64)
```

per ognuna delle sequenze SS_i il programma deve stampare i numeri che definiscono la sequenza, il valore minimo e massimo presente nella sequenza, il valore medio dei numeri che definiscono la sequenza.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run sequenze.go
Inserisci una sequenza di numeri (uno per riga); premi due volte invio per
terminare.
1
1
1
f
NaN... il valore non verrà considerato!
2
2
2
10

Sequenza S0:
[1 1 1 2 2 2 10]
Minimo in S0: 1
Massimo in S0: 10
Valore medio in S0: 2.7142857142857144

Sequenza S1:
[2 4 16 32 64 640 0.5 0.25 0.25 0.125 0.0625 0.15625]
Minimo in S1: 0.0625
Massimo in S1: 640
Valore medio in S1: 63.278645833333336

Sequenza S2:
[1 1 2 10]
Minimo in S2: 1
Massimo in S2: 10
Valore medio in S2: 3.5

Sequenza S3:
[1 1 6191 5175 4679 450 -6423 -8561 -650 7910 1340 4919 2 10]
Minimo in S3: -8561
Massimo in S3: 7910
Valore medio in S3: 1074.5714285714287
```

SUGGERIMENTO

Si utilizzi uno `Scanner` per gestire la terminazione della fase di inserimento dei numeri (l'inserimento dei numeri termina alla lettura di una riga vuota). Il dato inserito per ciascuna riga deve essere opportunamente convertito utilizzando le funzioni del package `strconv`. Se la conversione non va a buon fine, il dato viene scartato senza interrompere l'esecuzione del programma (cfr. "Esempio di funzionamento").

Esercizio 4 – Analisi di codice

```
func main() {

	const DIMENSIONE = 10

	var a []int

	for i:=0; i<DIMENSIONE; i++ {
		a = append([]int{i}, a...)
	}

	fmt.Println(a)

	b := make([]int, DIMENSIONE)

	copy(b, a[DIMENSIONE/2:])

	fmt.Println(b)

	c := make([]int, DIMENSIONE)

	for i:=0; i<DIMENSIONE; i+=2 {
		copy(c[DIMENSIONE-2-i:], a[i:i+2])
	}

	fmt.Println(c)
}
```

Esercizio 5 – Utilizzo della funzione copy – Copia di slice

Creare un programma che:

1) Legga un intero `n`
2) Crei due slice `sl1` e `sl2` entrambe di dimensione `n`
3) Inizializzi `sl1` con i numeri che vanno da `0` a `n-1`
4) Inizializzi `sl2` con i numeri che vanno da `n` a `2n-1`
5) Crei una terza slice `sl3` di dimensione `n`
6) Usi la funzione copy per copiare parte dei valori di `sl1` e `sl2` in `sl3`, in modo tale che: $sl3 = sl1_0, sl1_1, \dots, sl1_{n/2}, sl2_{n/2}, sl2_{n/2+1}, \dots, sl2_{2n-1}$

7) Stampi i valori contenuti in `sl3`

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run copia.go
5
[0 1 2 3 4] [5 6 7 8 9]
[0 1 7 8 9]
$ go run copia.go
10
[0 1 2 3 4 5 6 7 8 9] [10 11 12 13 14 15 16 17 18 19]
[0 1 2 3 4 15 16 17 18 19]
```

Esercizio 6 – Utilizzo della funzione copy – Spostamento degli elementi di un array/slice in avanti e all'indietro.

Scrivete un programma che riceve in input da linea di comando due interi `n` e `k`, con `k <= n`.

`n <= 26` rappresenta le dimensioni di un array `arr` di caratteri inizializzato con valori `'a' + 0, 'a' + 1, ..., 'a'+n-1`.

`2k` rappresenta il numero di volte che una variante dell'array `arr` deve essere stampato dal programma.

In particolare:

1) Sia `arr'` l'array da stampare l'i-esima volta, `0 <= i < k`, l'elemento `arr[h]` nell'array origianale è diventanto l'elemento `arr'[(h+i+1)%n]`.
2) Sia `arr''` l'array da stampare la j-esima volta, `0 <= j < k`, l'elemento `arr[h]` nell'array origianale è diventanto l'elemento `arr''[(h-j-1)%n]`.
   Si ricordi che: `-a % n = (n - a) % n` per `0 <= a <= n`.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run spostamento.go 4 4
Array di partenza:
a b c d

Stampa 0-esima (i=0): d a b c
Stampa 1-esima (i=1): c d a b
Stampa 2-esima (i=2): b c d a
Stampa 3-esima (i=3): a b c d

Stampa 0-esima (j=0): b c d a
Stampa 1-esima (j=1): c d a b
Stampa 2-esima (j=2): d a b c
Stampa 3-esima (j=3): a b c d
```

Esercizio 7 – Analisi di codice

Si consideri il seguente blocco di codice.

```
str := "ABD"
str[2] = 'C'
```

L'istruzione `str[2] = 'C'` genera l'errore "`cannot assign to str[2]`": una volta creata una stringa, non è possibile modificarne il contenuto.

Per cambiare `"ABD"` in `"ABC"` è possibile procedere, per esempio, in uno dei seguenti modi:

a)

```
strFrom := "ABD"
```

```
sl := []rune(strFrom)
/* la stringa strFrom è convertita in una slice sl di rune che ha una lunghezza pari
al numero di caratteri Unicode presenti in strFrom */
sl[2] = 'C'
strTo := string(sl)
/* strFrom == "ABD" e strTo == "ABC" */
```

b)

```
strFrom := "ABD"
strTo := strFrom[:2] + string('C')
/* strFrom == "ABD" e strTo == "ABC" */
```

c)

```
strFrom := "ABD"
//strTo := string(append([]byte(strFrom[:2]), []byte("C")...))
strTo := string(append([]byte(strFrom[:2]), "C"...))
/* una stringa, in questo caso "C", può essere "appended" direttamente ad una slice
di byte */
/* strFrom == "ABD" e strTo == "ABC" */
```

Esercizio 8 – Ordinamento con Selection Sort

Scrivere un programma che chieda all'utente di inserire una stringa `strDisordinata`, e quindi stampi una nuova stringa `strOrdinata` in cui i caratteri trovati in `strDisordinata`, esclusi i caratteri che rappresentano degli spazi, sono ordinati in senso non decrescente rispetto al codice Unicode ad essi associato.

Per l'ordinamento utilizzare l'algoritmo *Selection Sort*:

```
funzione SelectionSort(slice) {
  n = len(slice)
  for i=0; i<n; i++ {
    min = trova il numero di valore minimo all'interno della slice[i:n]
    indice = memorizza l'indice che min ha all'interno della slice[i:n]
    scambia i valori di slice[i] e slice[indice] in modo che il minimo finisca in testa
  }
}
```

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
> go run ordinaStringa.go
Inserisci la stringa da riordinare: Viva la Mamma
Stringa Ordinata: MVaaaailmmv
```

SUGGERIMENTO

Si dichiari la variabile `strOrdinata` di tipo stringa, inizializzandola con i caratteri presenti in `strDisordinata`, esclusi i caratteri che rappresentano degli spazi.

Si faccia riferimento alla modalità a) descritta nell'Esercizio 7 e si applichi l'algoritmo di ordinamento Selection Sort.

Esercizio 9 –Ordinamento di stringhe per inserzione

Scrivere un programma che chieda all'utente di inserire una stringa `strDisordinata`, e quindi stampi una nuova stringa `strOrdinata` in cui i caratteri trovati in `strDisordinata`, esclusi i caratteri che rappresentano degli spazi, sono ordinati in senso non decrescente rispetto al codice Unicode ad essi associato.

Per ordinare i carattteri in `strDisordinata`, utilizzate un algoritmo di ordinamento per inserzione. Il funzionamento dell'algoritmo è il seguente.

Si indichi con **`chDis(i)`** il carattere i-esimo di `strDisordinata`. Altrimenti detto:

`strDisordinata == "chDis(0)chDis(1)...chDis(len(strDisordinata)-1)".`

Innanzitutto, si dichiari la variabile `strOrdinata` di tipo stringa. Inizialmente, `strOrdinata` è una stringa vuota. Si considerino (con un ciclo) **tutti i caratteri `chDis(i)`** in `strDisordinata` e, per ogni carattere, si proceda come segue:

- se **`i=0`** (si sta considerando `chDis(0)`), `chDis(i)` viene inserito come primo carattere in `strOrdinata`.
- se **`i>0`**, si supponga che `strOrdinata` contenga i caratteri `chDis(j), j < i,` considerati fino all'iterazione precedente, ordinati in senso non decrescente rispetto al codice Unicode ad essi associato. `chDis(i)` deve essere inserito in strOrdinata prima del primo carattere associato ad un codice Unicode maggiore del codice unicode associato a `chDis(i)`.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
> go run ordinaStringa.go
9
```

```
Inserisci la stringa da riordinare: Viva la Mamma

Stringa Ordinata: MVaaaailmmv
```

SUGGERIMENTO

Si dichiari la variabile strOrdinata di tipo stringa. Inizialmente, strOrdinata è una stringa vuota.

Si faccia riferimento alla modalità b) descritta nell'Esercizio 7 e si applichi l'algoritmo di ordinamento per inserzione descritto nel testo dell'esercizio.

Lab08 – Mappe

Esercizio 1 – Analisi di codice

```go
func main() {

  m := make(map[string]int)

  for _, s := range []string{"questo", "è", "un", "test"} {
     m[s] = len([]rune(s))
  }

  for k, v := range m {
     fmt.Println(k, "->", v)
  }

}
```

Esercizio 2 – Analisi di codice

```go
func main() {

  mappa := make(map[string]int)
  // equivalente a: mappa := map[string]int{}


  mappa["A"] = 10
  mappa["B"] -= 5
  mappa["D"] = mappa["D"] + 5

  if v, ok := mappa["B"]; ok {
     fmt.Printf("B è presente con valore %d\n", v)
  } else {
     fmt.Print("B non è presente\n")
  }
```

```
  if v, ok := mappa["C"]; ok {
     fmt.Printf("C è presente con valore %d\n", v)
  } else {
     fmt.Print("C non è presente\n")
  }

  if v, ok := mappa["C"]; ok {
     fmt.Printf("C è presente con valore %d\n", v)
  } else {
     fmt.Print("C non è presente\n")
  }

  delete(mappa, "B")

  for k, v := range mappa {
     fmt.Printf("chiave %s valore %d\n", k, v)
  }
}
```

Esercizio 3 – Istogramma di lettere

PARTE I

Scrivere un programma che legga da tastiera un testo (su più righe) e stampi un istogramma che rappresenti il numero di occorrenze di ogni lettera all'interno del testo. L'istogramma prodotto deve essere *case sensitive*, ovvero distinguere tra lettere minuscole e maiuscole. Usate degli asterischi (*) per rappresentare la dimensione delle barre dell'istogramma.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go istogramma.go
```

Questo è un

ESEMPIO di testo!

```
Occorrenze:
```

```
I: *
O: *
Q: *
E: **
S: *
M: *
P: *
s: **
n: *
d: *
u: **
e: **
t: ***
o: **
è: *
i: *
```

SUGGERIMENTO

Per contare quante volte una lettera è stata ripetuta utilizzate una tipo di dato map[rune]int

PARTE II

Modificare il programma in modo che sia *case insensitive* numero di occorrenze di ogni carattere alfabetico contenuto nel testo.

Esempio di funzionamento (in grassetto l'output inserito dall'utente)

```
$ go run istogramma.go
Questo è un
ESEMPIO di testo!
```

Occorrenze:

q: *

u: **

n: *

p: *

i: **

d: *

e: ****

s: ***

t: ***

o: ***

è: *

m: *