Lab11 – Esercizi vari

Esercizio 1 - Terne pitagoriche

In un triangolo rettangolo di cateti `a`, `b` e di ipotenusa `c` si ha: $a^2 + b^2 = c^2$. Scrivere un programma che legga da riga di comando un intero `n > 0` e stampi tutti i numeri naturali inferiori a `n` che non possono rappresentare la lunghezza dell'ipotenusa di un triangolo rettangolo, numeri cioè il cui quadrato non può essere espresso come somma dei quadrati di due numeri naturali distinti e non nulli.

Esempio di funzionamento

```
$ go run terne.go 30
1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 14, 16, 18, 19, 21, 22, 23, 24, 27, 28
```

Esercizio 2 - Comandi

Defirnire le seguenti funzioni il cui input è costituito da valori interi positivi o nulli:

`func somma(n, m uint) uint` /* funzione che restituisce la somma di `n` e `m` */

`func moltiplica(n, m uint) uint` /* funzione che restituisce il risultato della moltiplicazione di `n` e `m` */

`func eleva(b, exp uint) uint` /* funzione che restituisce la potenza b^{exp} */

Scrivere un programma che legga una sequenza di comandi da standard input (uno per riga). La fase di inserimento dei comandi termina alla lettura di una riga vuota.

I comandi hanno il seguente `formato` *significato*:

`S int1 int2` *Comando che richiede di stampare il risultato della somma* `int1+int2`

`M int1 int2` *Comando che richiede di stampare il risultato della moltiplicazione* `int1*int2`

`P int1 int2` *Comando che richiede di stampare il risultato dell'elevamento a potenza* $int1^{int2}$

Una volta terminata la fase di inserimento dei comandi, il programma li interpreta in sequenza, eseguendo

ogni comando e documentandone l'avvenuta esecuzione. Ogni comando viene eseguito richiamando la corrispondente funzione.

Esempio di funzionamento

```
$ go run operazioni.go
P 2 4
S 7 9

Elevo a potenza: 2^4 = 16
Sommo: 7+9 = 16
```

Contenuto di `input.txt`:

```
------------------------------------------------------------------------
P 2 4
S 7 9
S 5 3
M 13 7
S 1 1
M 5 3
S 19 0
P 17 3
M 13 0
M 0 0
P 65476326 0
P 657826487 1
------------------------------------------------------------------------
```

```
$ go run operazioni.go < input.txt
Elevo a potenza: 2^4 = 16
Sommo: 7+9 = 16
Sommo: 5+3 = 8
Moltiplico: 13*7 = 91
Sommo: 1+1 = 2
Moltiplico: 5*3 = 15
Sommo: 19+0 = 19
Elevo a potenza: 17^3 = 4913
Moltiplico: 13*0 = 0
Moltiplico: 0*0 = 0
Elevo a potenza: 65476326^0 = 1
Elevo a potenza: 657826487^1 = 657826487
```

Esercizio 3 - Stampa alternata

Scrivere un programma che legga da riga di comando **n** valori interi memorizzandoli in un array/slice, e stampi il primo elemento del array/slice, poi l'ultimo, poi il secondo, poi il penultimo, …. fino a stampare tutti i valori letti. Se **n=0** il programma stampa: "`Nessun valore!`".

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

```
>go run stampaAlternata.go 11
1 2 3 4 5 6 7 8 9 10 11
Stampa alternata:
1 11 2 10 3 9 4 8 5 7 6
>go run stampaAlternata.go 12
-1 2 -3 4 -5 6 -7 8 -9 10 -11 12
Stampa alternata:
-1 12 2 -11 -3 10 4 -9 -5 8 6 -7
>go run stampaAlternata.go 1
1
Stampa alternata:
1
>go run stampaAlternata.go 2
-1
-2
Stampa alternata:
-1 -2
>go run stampaAlternata.go 0
Nessun valore!
>go run stampaAlternata.go 9
1 -2 1 -2 1 -2 1 -2 1
Stampa alternata:
1 1 -2 -2 1 1 -2 -2 1
```

Esercizio 4 - Vicini più vicini

Scrivere un programma che legga:

- un valore intero `r >= 0` da riga di comando;
- una sequenza di numeri reali (ovvero di tipo float64) da standard input (i numeri sono inseriti su un'unica riga, due numeri consecutivi sono separati da un singolo carattere spazo ' ', la lunghezza della sequenza di numeri non è nota a priori).

Il programma deve stampare a video:

- i numeri arrotondati alla `r`-esima cifra decimale (un unico numero per riga);
- la coppia di numeri contigui che, dopo l'arrotondamento, è caratterizzata dalla minore distanza (dove la distanza tra due numeri `x` e `y` è definita dal valore `|x-y|`, ossia il valore assoluto della differenza tra `x` e `y`). L'ultimo elemento della sequenza è da considerarsi contiguo al primo. Ad esempio, se la sequenza inserita fosse: 1 2 3 4 5, le coppie di numeri contigui sarebbero: 1 e 2, 2 e 3, 3 e 4, 4 e 5, 5 e 1.

Esempio di funzionamento

Conenuto di `numeri.txt`:

```
8.2346 6.9483 3.1710 9.5022 0.3445 4.3874 3.8156
```

```
>go run vicinipiuvicini.go 2 < numeri.txt
8.23
6.95
3.17
9.5
0.34
4.39
3.82
Vicini più vicini: 4.39 e 3.82
```

Esercizio 5 - Il codice di Vigénère

Un sistema di cifratura molto diffuso fin dal XVI secolo è il cosiddetto *codice di Vigénère*, una variante polialfabetica del cifrario di Cesare.

Si supponga di avere un TESTO_IN_CHIARO, costituito semplicemente da una sequenza di caratteri alfabetici (si consideri l'insieme C di tutti caratteri che corrispondono a lettere dell'alfabeto inglese). Per applicare il codice di Vigénère, occorre anche avere una CHIAVE_DI_CIFRATURA, spesso chiamata 'verme'.

Il TESTO_IN_CHIARO (privato di tutti i caratteri non alfabetici) e il verme vengono scritti uno sopra l'altro (se necessario, il verme viene ripetuto più volte e/o troncato, in modo che le due sequenze di caratteri alfabetici abbiano la stessa lunghezza). Quindi i due testi vengono sommati lettera per lettera. In pratica, questo corrisponde a identificare ogni lettera dell'alfabeto con un numero fra 0 e 25, e nell'effettuare le somme modulo 26.

Ad esempio, se il TESTO_IN_CHIARO fosse "ARRIVANO!! I RINFORZI!" e il verme fosse proprio la parola "VERME":

- Si eliminano tutti i caratteri non alfabetici dal TESTO_IN_CHIARO, ottenendo:

ARRIVANOIRINFORZI

- Si ripete la parola "VERME", eventualmente troncandola, fino ad avere lo stesso numero di caratteri del testo da convertire:

VERMEVERMEVERMEVE

- Si sommano le lettere corrispondenti:

A	R	R	I	V	A	N	O	I	R	I	N	F	O	R	Z	I
								+								
V	E	R	M	E	V	E	R	M	E	V	E	R	M	E	V	E

- Si ottiene quindi il risultato:

V	V	I	U	Z	V	R	F	U	V	D	R	W	A	V	U	M

- Si ristampa il risultato inserendo i caratteri non alfabetici che erano stati rimossi:

VVIUZVRF U VDRWAVUM

Per spiegare meglio, notate che nell'esempio sopra:

A+V=V (essendo A la 0-esima lettera e V la 21-esima lettera, A+V=0+21=21, 21%26=21=V),

...

I+V=D (I è la 8-esima lettera e V la 21-esima, I+V=8+21=29, 29%26=3=D)

...

Notate inoltre che lo stesso algoritmo si può usare anche per decifrare un TESTO_CIFRATO: è sufficiente sostituire al verme usato il suo "inverso" (sostituendo ad ogni A una A, ad ogni B una Z, ad ogni C una Y, ad ogni D una X, ...).

Scrivere un programma che legga da tastiera un verme, legga poi il TESTO_IN_CHIARO e infine stampi il TESTO_CIFRATO. Notate che il programma crittografa solo i caratteri alfabetici; i caratteri non alfabetici (ad.es. spazi, virgole, etc..) devono rimanere tali e quali. In aggiunta, le lettere minuscule rimangono minuscole, mentre le lettere maiuscole rimangono maiuscole.

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

```
>go run vigenere.go
VERME: VERME
TESTO IN CHIARO: Arrivano!! I RINFORZI!
TESTO CIFRATO: Vviuzvfr!! U VDRWAVUM!

>go run vigenere.go
VERME: FWJOW
TESTO IN CHIARO: VVIUZVRF!!! U vdrwavum
TESTO CIFRATO: ARRIVANO!!! I rinforzi
```

Esercizio 6 - Il crivello di Eratostene

Scrivere un programma che legge da riga di comando un intero `n`, calcoli con l'algoritmo noto con il nome di "crivello di Eratostene" i numeri primi minori o uguali a `n`, stampi i numeri primi calcolati.

Il funzionamento dell'algoritmo è il seguente.

Si considera l'elenco di tutti i numeri naturali a partire da 2 fino `n`, tale elenco è detto "setaccio".

Si considera il primo elemento `p` presente nel setaccio.

Si cancellano (setacciano) tutti i multipli di `p` (`2p, 3p, 4p, ...`) presenti nel setaccio (i multipli di `p`

devono essere minori o uguali a n).

Si considera il primo numero maggiore di p ancora presente nel setaccio e si ripete l'operazione precedente (nel caso in cui non esistano nel setaccio ulteriori numeri maggiori di p, l'algoritmo termina).

I numeri che restano nel setaccio sono i numeri primi minori o uguali a n.

Esempio di funzionamento

```
>go run eratostene.go 20
2 3 5 7 11 13 17 19
>go run eratostene.go 100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
>go run eratostene.go 41
2 3 5 7 11 13 17 19 23 29 31 37 41
```

Esercizio 7 - Domino

Scrivere un programma che legga:

- una parola p da riga di comando;
- una sequenza di parole da standard input (la sequenza non ha dimensione definita; le parole sono inserite ciascuna su una riga diversa; la fase di inserimento delle parole termina alla lettura di una riga vuota);

e crei un "domino di parole" che si "agganciano" in base al carattere finale/iniziale (ovvero una sequenza di parole "agganciate" perchè l'ultimo carattere di una parola è uguale al primo carattere della successiva) utilizzando l'algoritmo di seguito descritto.

Sia IN = [P_0, P_1, P_2, ..., P_{n-1}, P_n] la sequenza di parole lette da standard input.

Sia DOMINO la sequenza di parole che al termine del programma rappresenterà il "domino di parole" creato.

I passi dell'algoritmo sono i seguenti:

1. `OUT = [p]` /* inizializzazione */
2. Si scorre `IN` dall'inizio cercando la prima parola della sequenza che comincia con l'ultimo carattere dell'ultima parola inserita in `DOMINO` (ovvero della parola che è stata agganciata al domino al passo precedente). Se in `IN` non si trova una parola adatta, l'algoritmo termina; in caso contrario sia P_i la parola trovata, P_i viene rimossa da `IN` e viene aggiunta come ultima parola alla sequenza `DOMINO`.
3. Se `IN` è vuota l'algoritmo termina, altrimenti si ripete il passo 2.

L'algoritmo non fa distinzione tra caratteri maiuscoli e minuscoli.

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

Contenuto di `input.txt`:

```
--------------------------------------------------------------------------
Abba
Abracadabra
Martello
Telegiornale
Cartone
Vetro
Toronto
Violoncello
EVVIVA
Caffettiera
mio
amico
Cinema
festa
di
compleanno
Viviana
Natale
stop
--------------------------------------------------------------------------
```

```
>go run domino.go mamma < input.txt

Parole lette: [ABBA, ABRACADABRA, MARTELLO, TELEGIORNALE, CARTONE, VETRO,
TORONTO, VIOLONCELLO, EVVIVA, CAFFETTIERA, MIO, AMICO, CINEMA, FESTA, DI,
COMPLEANNO, VIVIANA, NATALE, STOP]
Parto con parola: MAMMA
Non trovo aggancio: stop!
Domino risultante: [MAMMA, ABBA, ABRACADABRA, AMICO]
Parole non utilizzate: [MARTELLO, TELEGIORNALE, CARTONE, VETRO, TORONTO,
VIOLONCELLO, EVVIVA, CAFFETTIERA, MIO, CINEMA, FESTA, DI, COMPLEANNO, VIVIANA,
NATALE, STOP]
```

Esercizio 8 - Chiocciole quadrate di numeri

Scrivere un programma che legga da riga di comando due interi `n >= 1` e `startNum` e stampi i numeri da `startNum` in poi, disponendoli secondo la forma di una chiocciola quadrata di dimensione `n x n`.

Esempio di funzionamento

```
>go run chiocciole.go 5 7
7       8       9       10      11
22      23      24      25      12
21      30      31      26      13
20      29      28      27      14
19      18      17      16      15


> go run chiocciole.go 10 1
1       2       3       4       5       6       7       8       9       10
36      37      38      39      40      41      42      43      44      11
35      64      65      66      67      68      69      70      45      12
34      63      84      85      86      87      88      71      46      13
33      62      83      96      97      98      89      72      47      14
32      61      82      95      100     99      90      73      48      15
31      60      81      94      93      92      91      74      49      16
30      59      80      79      78      77      76      75      50      17
29      58      57      56      55      54      53      52      51      18
28      27      26      25      24      23      22      21      20      19


>go run chiocciole.go 1 1321432
1321432
```

Esercizio 9 - Ricerca ricorsiva

Scrivere un programma che:

- legga una sequenza di `n` numeri interi passati come argomento da linea di comando;
- ordini i numeri in senso non decrescente ottenendo la sequenza ordinata di numeri `a = a(0),...,a(n-1)` dove i valori tra parentesi indicano gli indici di posizione del numero nella sequenza;
- chieda all'utente di inserire un numero `x` da standard input;
- verifichi la presenza del numero `x` nella sequenza `a(0),..., a(n-1)` tramite una funzione ricorsiva di tipo *"divide et impera"*, stampando la posizione della prima occorrenza di `x` nella sequenza se `x` è presente, oppure -1 se `x` non è presente.

La funzione ricorsiva di tipo *"divide et impera"* ricerca il numero `x` nella sequenza di numeri `a(0),...,a(n-1)` nel seguente modo:

- se la sequenza è vuota, restituisce -1
- se la sequenza è composta da un solo elemento `a(0)` e `a(0)==x`, la funzione ricorsiva restituisce il valore `0`; altrimenti restituisce `-1`;
- se la sequenza è composta da `n > 1` numeri, sia `k = n/2` (`k` il risultato della divisione tra gli interi `n` e 2); se `a(k)==x` la funzione ricorsiva restituisce il valore `k`; in caso contrario valuta i seguenti casi:
  - se `a(k)>x`, allora il numero va ricercato in modo ricorsivo nella sequenza `a(0),...,a(k-1)` ed il valore restituito dalla funzione ricorsiva sarà il risultato di tale ricerca;
  - se `a(k)<x` allora il numero va cercato in modo ricorsivo nella sequenza `a(k+1),..., a(n-1)`; sia `ris` il risultato di tale ricerca, il valore restituito dalla funzione ricorsiva sarà `ris + (k+1)` se `ris != -1`, `ris` altrimenti.

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

```
$ go run ricerca.go 3 7 5 -2 10 1
Sequenza ordinata: [-2 1 3 5 7 10]
Inserisci il valore intero da ricercare: 3
Posizione del valore ricercato: 2

$ go run ricerca.go 3 7 5 -2 10 1
Sequenza ordinata: [-2 1 3 5 7 10]
Inserisci il valore intero da ricercare: 4
Posizione del valore ricercato: -1
```

Esercizio 10 - Stampa ordinata delle coppie presenti in una mappa

Scrivere un programma che stampi le coppie presenti nella mappa

```
stringaNumero := map[string]int{"alpha": 34, "bravo": 56, "charlie": 23,
        "delta": 87, "echo": 56, "foxtrot": 12, "golf": 34, "hotel": 16,
        "india": 87, "juliet": 65, "kilo": 43, "lima": 98}
```

in ordine alfabetico rispetto al valore della chiave. Ai fini dell'ordinamento, si ricorda che il confronto lessicografico tra due stringhe può essere effettuato utilizzando la funzione `strings.Compare`.

Esempio di funzionamento

```
$ go run stampaOrdinata/stampaOrdinata.go
alpha: 34
bravo: 56
charlie: 23
delta: 87
echo: 56
foxtrot: 12
golf: 34
hotel: 16
india: 87
juliet: 65
kilo: 43
lima: 98
```

Esercizio 11 - Istogramma di lettere

Scrivere un programma che legga da standard input un testo e, per ogni carattere alfabetico contenuto nel testo (si consideri l'insieme C di tutti caratteri che corrispondono a lettere dell'alfabeto inglese), stampi a video una riga così definita:

`carattere_alfabetico -> numero_occorrenze`

dove `numero_occorrenze` è il numero di occorrenze di `carattere_alfabetico` nel testo letto.

Le righe del file di output devono rispettare l'ordine alfabetico in base ai caratteri che le caratterizzano. Scrivere due versioni del programma:

(1) una versione case sensitive in cui i caratteri maiuscoli sono considerati diversi da quelli minuscoli;

(2) una versione case insensitive in cui non c'è distizione tra caratteri maiuscoli e minuscoli (in questa versione il carattere che caratterizza ciascuna riga del file di output deve essere stampato in maiuscolo).

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

Contenuto di `input.txt`:

```
-------------------------------------------------------------------------------
```

```
A Carnevale,
Zorro si traveste da Chuck Norris.

Zorro firma con la sua spada una "z" sui vestiti dell'avversario.
Chuck Norris con un calcio rotante stampa in fronte nome,
cognome, indirizzo e C.A.P.
--------------------------------------------------------------------------------

> go run istogrammaLettere.go < input.txt > output.txt
```

Contenuto di `output.txt`:

```
--------------------------------------------------------------------------------
A: 2
C: 4
N: 2
P: 1
Z: 2
a: 16
c: 7
d: 4
e: 12
f: 2
g: 1
h: 2
i: 13
k: 2
l: 5
m: 4
n: 11
o: 16
p: 2
r: 16
s: 10
t: 8
u: 6
v: 5
z: 3
--------------------------------------------------------------------------------
```

Notate che, poiché il testo non contiene alcuna lettera 'B', la riga relativa a 'B' non viene stampata nel file di output; lo stesso vale per 'D', 'E', …

Esercizio 12 - Punti

Si consideri il package `punto` la cui definizione è di seguito riportata.

```
package punto

import "fmt"

var nextID uint64 = 0
```

```
type Punto struct {
  ID uint64      /* Identificativo univoco del punto */
  x, y float64   /* Coordinate del punto sul piano cartesiano */
}

// Crea un nuovo punto restituendone il puntatore.
// x e y sono inizializzati al valore di default.
func NewPunto(x, y float64) *Punto {
  p := &Punto{nextID, x, y}
  nextID++
  return p
}

func ID(p *Punto) uint64 {
  return p.ID
}

func X(p *Punto) float64 {
  return p.x
}

func Y(p *Punto) float64 {
  return p.y
}

func SetX(p *Punto, X float64) {
  p.x = X
}

func SetY(p *Punto, Y float64) {
  p.y = Y
}

func String(p *Punto) string {
  s := fmt.Sprintf("P(ID = %v, X = %6.3f, Y = %6.3f)",  p.ID, p.x, p.y);
  return s
}
```

Si noti che la variabile globale `punto.nextID` (non visibile al di fuori del package) è utilizzata per assegnare in modo automatico un identificativo univoco (e progressivo) a ciascun nuovo/a oggetto/istanza del tipo `Punto`, l'istanza deve però essere creata tramite la funzione `func NewPunto(x, y float64) *Punto`.

Scrivere un programma che, utilizzando le funzionalità messe a disposizione dal package `punto`, permetta di creare, modificare e interrograre un elenco di punti appartente al piano cartesiano.

In particolare, il programma deve leggere una sequenza di comandi da standard input (uno per riga). La fase di inserimento dei comandi termina alla lettura di una riga vuota.

I comandi hanno il seguente `formato`/*significato*:

`INS X_P Y_P`	*Comando che richiede di inserire nell'elenco un nuovo punto con coordinate x=X_P e y=Y_P, dove X_P e Y_P sono due numeri reali.*

`DEL ID_P`	*Comando che richiede di eliminare nell'elenco il punto con identificativo ID_P. Se il punto non è presente in elenco, il comando non ha effetto e viene stampato in output il seguente messaggio:* *"Il comando non può essere eseguito. Il punto con identificativo ID_P non è presente in elenco."*
`SETX ID_P X_P`	*Comando che richiede di modificare il valore della coordinata x del punto con identificativo ID_P. Il valore della coordinata x deve essere impostato a X_P. Se il punto non è presente in elenco, il comando non ha effetto e viene stampato in output il seguente messaggio:* *"Il comando non può essere eseguito. Il punto con identificativo ID_P non è presente in elenco."*
`SETY ID_P Y_P`	*Comando che richiede di modificare il valore della coordinata y del punto con identificativo ID_P. Il valore della coordinata y deve essere impostato a Y_P. Se il punto non è presente in elenco, il comando non ha effetto e viene stampato in output il seguente messaggio:* *"Il comando non può essere eseguito. Il punto con identificativo ID_P non è presente in elenco."*
`Q Q_ID`	*Comando che richiede di stampare a video tutti i punti presenti in elenco che appartengono al quadrante Q_ID del piano cartesiano (1 <= Q_ID <= 4). L'ordine in cui vengono stampati i punti non è soggetto ad alcun vincolo.*
`PRINT`	*Comando che richiede di stampare a video tutti i punti presenti in elenco. L'ordine in cui vengono stampati i punti non è soggetto ad alcun vincolo.*

Una volta terminata la fase di inserimento dei comandi, il programma li interpreta in sequenza, eseguendo ogni comando e documentandone l'avvenuta esecuzione.

SUGGERIMENTI

- Lo scheletro del programma potrebbe essere il seguente:

```
func main() {

  fmt.Println("Inserisci i comandi...")

  comandi := leggiComandi() /* comandi è una var di tipo []string */

  fmt.Println("Lettura dei comandi terminata!\n")

  fmt.Println("Esecuzione dei comandi:")

  var elenco []*punto.Punto

  for i := 0; i < len(comandi); i++ {
    s := comandi[i]

    var c string
    fmt.Sscan(s, &c)

    switch c {
```

```
    case "INS":
      /* lettura dei parametri del comando */
      var x, y float64

      fmt.Sscanf(s, "INS%f%f", &x, &y)

      fmt.Println("COMANDO:", "I", x, y)
      /* lettura dei parametri del comando - end */

      elenco = inserisci(elenco, x, y)

    case "PRINT":
      /* lettura dei parametri del comando */
      var t string

      fmt.Sscanf(s, "PRINT%s", &t)

      fmt.Println("COMANDO:", "PRINT", t)
      /* lettura dei parametri del comando - end */

       stampa(elenco)

    default:
      fmt.Println("COMANDO:", s)
      fmt.Println("Comando sconosciuto; il comando è stato ignorato.")

    }

    fmt.Println()

  }

}
```

- Notate che, una volta letto, il comando viene ristampato a video. Questo permette di controllare se la lettura del comando (ed in particolare dei suoi parametri) è stata effettuata correttamente.
- Il codice del blocco `switch {...}` può essere esteso incrementalmente, aggiungendo di volta in volta un nuovo case per gestire l'esecuzione di uno dei comandi non ancora considerati. Nel momento in cui è stato attestato il corretto funzionamento del codice aggiunto, si può procedere nel considerare un nuovo comando la cui esecuzione non è ancora stata gestita.
- Al fine di migliorare la leggibilità e manutenibilità del codice, sarebbe opportuno definire per ognuno dei comandi una funzione che ne gestisca l'esecuzione (da richiamare nel case corrispondente), evitando quindi di scrivere tutto il codice nel `main`.
- La variabile `elenco` potrebbe anche essere definita come:

  `elenco:=map[unit64]*punto.Punto`

  Se lo ritenete opportuno, utilizzate questa definizione alternativa della variabile elenco, modificando opportunamente `func main() {...}`.

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

Contenuto di `comandi.txt`:

```
------------------------------------------------------------------------
INS 4.5343 5.89
INS 7.3 6.89
INS 43 -5.8
DEL 0
Q 1
SETX 2 -4
Q 3
FAIL
PRINT
------------------------------------------------------------------------
```

```
$ go run elenco.go < comandi.txt
Inserisci i comandi:
Lettura dei comandi terminata!

Esecuzione dei comandi:
COMANDO: INS 4.5343 5.89
Nuovo punto inserito: P(id = 0, X =  4.534, Y =  5.890)

COMANDO: INS 7.3 6.89
Nuovo punto inserito: P(id = 1, X =  7.300, Y =  6.890)

COMANDO: INS 43 -5.8
Nuovo punto inserito: P(id = 2, X = 43.000, Y = -5.800)

COMANDO: DEL 0
Punto eliminato: P(id = 0, X =  4.534, Y =  5.890)

COMANDO: Q 1
Punti nel quadrante 1:
P(id = 1, X =  7.300, Y =  6.890)

COMANDO: SETX 2 -4
Cambiata coordinata X punto ID 2: P(id = 2, X = -4.000, Y = -5.800)

COMANDO: Q 3
Punti nel quadrante 3:
P(id = 2, X = -4.000, Y = -5.800)

COMANDO: FAIL
Comando sconosciuto; il comando è stato ignorato.
```

```
COMANDO: PRINT
Punti nell'elenco:
P(id = 1, X =  7.300, Y =  6.890)
P(id = 2, X = -4.000, Y = -5.800)
```

Esercizio 13 - Confronti tra date

Scrivere un programma che:

1. Legga un intero n da riga di comando
2. Generi n date in modo casuale
3. Confronti le date generate e stampi la data maggiore e quella minore
4. Calcoli e stampi la differenza tra la data maggiore e quella minore

SUGGERIMENTO

Per creare una data, si usi la funzione Date del package `time`. L'istruzione necessaria per creare la data 14/01/2019 è

```
data := time.Date(2019, 01, 14, 0, 0, 0, 0, 0, time.Local)
```

I primi 3 interi di questa funzione sono anno, mese e giorno. Si usino le funzioni del package `math/rand` per generare in modo casuale i valori interi di anno, mese e giorno, in modo da passarli successivamente come parametri della funzione `Date`.

Per confrontare le date (quale data viene prima e quale viene dopo), potete usare le funzioni `Before` e `After`:

```
d1.Before(d2)  // Controlla se la data contenuta in d1 precede la data contenuta in d2
d1.After(d2)   // Controlla se la data contenuta in d1 segue la data contenuta in d2
```

I dettagli di queste funzioni sono riportati nella documentazione del package time (https://golang.org/pkg/time)

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

```
$ go run date/date.go 5
Date generate: [2000-02-29 00:00:00 +0100 CET 2033-07-20 00:00:00 +0200
CEST 2016-11-24 00:00:00 +0100 CET 2037-10-09 00:00:00 +0200 CEST 2023-
02-26 00:00:00 +0100 CET]
Data maggiore:  2037-10-09 00:00:00 +0200 CEST
Data minore:  2000-02-29 00:00:00 +0100 CET
Diffrenza in ore/minuti/secondi:  329687h0m0s
Diffrenza in secondi:  1.1868732e+09
```