# Lab05 – Rappresentazione dell'informazione/Stringhe/Iterazione

## Esempio 1 - Indovina l'algoritmo

Cosa stampa il seguente codice?

```
func main() {

	var x int
	var y int = 'a'
	x = 'A'
	fmt.Print(x, " ")
	for x := 1; x < 10; x++ {
		fmt.Print(x+y, " ")
	}
	fmt.Println()
}
```

## Esempio 2 - Indovina l'algoritmo

Cosa stampa il seguente codice?

```
func main() {

  var x int = 10
  var y float64 = 2.5
  t := 100

  t, z, k := x/int(y), float64(x)/y, t*2

  fmt.Println(t, z, k)
}
```

## Esempio 3 - Trova gli errori

Trova e correggi gli errori in modo che il codice compili

```
func main() {
  var a int = 1<<2
  var b float64 = 12.5
  var e
  var c int

  c := a + b
  d = a/b

  fmt.Println(a, b, c, d, f)
}
```

## Esercizio 1 - Integer Overflow/Underflow

Scrivere un programma che riceve in ingresso da standard input un valore intero e lo salva in una variabile di tipo `int` (che chiameremo `n`). Create quindi due variabili `n64` (di tipo `int64`) e `n32` (di tipo `int32`), e copiateci il valore di `n`. A questo punto incrementate iterativamente `n`, `n32` e `n64` e stampate il loro valore dopo ogni incremento.

In questo modo potrete controllare cosa succede quando il valore contenuto in variabili di tipo `int`, `int32`, `int64` non è più memorizzabile nelle variabili stesse. Riuscite a spiegarlo? Cosa succede se si modifica lo stesso programma al fine di decrementare la variabile anziché incrementarla?

Che cosa succede se si ripete l'esercizio precedente considerando valori `float32`, `float64`?

**SUGGERIMENTO:**

Partite da un valore pari a `2147483640` (questo valore è pari $(2^{31} - 8)$!!!) e fermate il ciclo appena una delle seguenti condizioni diventa vera (notate che nessuna di queste condizioni si dovrebbe avverare se stessimo parlando di numeri appartenenti all'insieme dei numeri interi - indicato in matematica con il simbolo $\mathbb{Z}$ - o dei numeri reali - indicato in matematica con il simbolo $\mathbb{R}$):

```
(n+1) < n

(n32+1) < n32

(n64+1) < n64
```

Osservate il seguente esempio di esecuzione (in grassetto il numero inserito dall'utente). Notate cosa succede alla variabile di tipo `int32`

**Esempio di esecuzione (in grassetto il valore inserito dall'utente):**

```
>go run esercizio1.go

Inserisci un valore intero:

2147483640

+1

int64 = 2147483641

int32 = 2147483641

int = 2147483641

+1

int64 = 2147483642

int32 = 2147483642

int = 2147483642

+1

int64 = 2147483643

int32 = 2147483643

int = 2147483643

+1

int64 = 2147483644

int32 = 2147483644

int = 2147483644

+1
```

```
int64 = 2147483645
int32 = 2147483645
int = 2147483645
+1
int64 = 2147483646
int32 = 2147483646
int = 2147483646
+1
int64 = 2147483647
int32 = 2147483647
int = 2147483647
+1
int64 = 2147483648
URKA32! int32 overflow: int32 = -2147483648
int = 2147483648
CONDIZIONE NON VERIFICATA PER int32
```

**Nell'esempio sopra in pratica è successo che la variabile di tipo int32, dopo aver assunto il massimo valore positivo che può contenere - ovvero il valore $2^{31}-1$ – al primo incremento "ricomincia il giro" partendo dal minimo valore che può assumere, ovvero il valore $(-2^{31})$.**

Ora partite dal valore `9223372036854775800` che si ottiene dal calcolo $(2^{63} - 8)$, e guardate cosa succede (inserisco i commenti al fianco del risultato)

```
>go run esercizio1.go
Inserisci un valore intero:
9223372036854775800
+1
```

```
int64 = 9223372036854775801
int32 = -7              //la variabile di tipo int32 parte già da un valore
                        //negativo perchè non può contenere il valore
                        //inserito dall'utente!
int = 9223372036854775801
+1
int64 = 9223372036854775802
int32 = -6
int = 9223372036854775802
+1
int64 = 9223372036854775803
int32 = -5
int = 9223372036854775803
+1
int64 = 9223372036854775804
int32 = -4
int = 9223372036854775804
+1
int64 = 9223372036854775805
int32 = -3
int = 9223372036854775805
+1
int64 = 9223372036854775806
int32 = -2
int = 9223372036854775806
+1
```

```
int64 = 9223372036854775807          //la variabile int64 assume il massimo
                                     //valore possibile per una variabile di
                                     //questo tipo: è il valore 2^63-1

int32 = -1

int = 9223372036854775807

+1

URKA64! int64 overflow: int64 = -9223372036854775808

                                     //la variabile int64 va in overflow e
                                     //riparte dal valore più piccolo che può
                                     //assumere, ovvero il valore: -2^63

int32 = 0

URKA! int overflow: int = -9223372036854775808

CONDIZIONE NON VERIFICATA PER int, int64
```

## Esercizio 2 – Overflow/Underflow

```
uint8       the set of all unsigned  8-bit integers (0 to 255)
uint16      the set of all unsigned 16-bit integers (0 to 65535)
uint32      the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8        the set of all signed  8-bit integers (-128 to 127)
int16       the set of all signed 16-bit integers (-32768 to 32767)
int32       the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       the set of all signed 64-bit integers (-9223372036854775808 to 9223372036854775807)

float32     the set of all IEEE-754 32-bit floating-point numbers
float64     the set of all IEEE-754 64-bit floating-point numbers

complex64   the set of all complex numbers with float32 real and imaginary parts
complex128  the set of all complex numbers with float64 real and imaginary parts

byte        alias for uint8
rune        alias for int32
```

Individuare anche per ognuno dei tipi numerici:

- quale sia l'ordine di grandezza del più grande valore positivo e del più piccolo valore negativo non accettati.

Tenete presente che i massimi valori assumibili dai tipi numerici basilari sono salvati nelle costanti del package `math`:

Floating-point limit values. Max is the largest finite value representable by the type. SmallestNonzero is the smallest positive, non-zero value representable by the type.

```
const (
        MaxFloat32             = 3.40282346638528859811704183484516925440e+38  // 2**127 * (2**24 - 1) / 2**23
        SmallestNonzeroFloat32 = 1.40129846432481707092372958328991613128Oe-45 // 1 / 2**(127 - 1 + 23)

        MaxFloat64             = 1.79769313486231570814527423731704356798le+308 // 2**1023 * (2**53 - 1) / 2**52
        SmallestNonzeroFloat64 = 4.94065645841246544176568792868221372365le-324 // 1 / 2**(1023 - 1 + 52)
)
```

Integer limit values.

```
const (
        MaxInt8   = 1<<7 - 1
        MinInt8   = -1 << 7
        MaxInt16  = 1<<15 - 1
        MinInt16  = -1 << 15
        MaxInt32  = 1<<31 - 1
        MinInt32  = -1 << 31
        MaxInt64  = 1<<63 - 1
        MinInt64  = -1 << 63
        MaxUint8  = 1<<8 - 1
        MaxUint16 = 1<<16 - 1
        MaxUint32 = 1<<32 - 1
        MaxUint64 = 1<<64 - 1
)
```

## Esercizio 3 - Uguaglianza tra valori `float32` e `float64`

Scrivete un programma che riceve in input un valore `x (float64)`, e calcola la sua radice quadrata che salva in un'altra variabile `y (float64)`.

Il programma controlla se `y*y` e' uguale a `x` e in tal caso stampa la stringa "`x uguale a y*y`"; se invece `y*y` è diverso da `x`, stampa la stringa "`x diverso da y*y`".

**Esempi d'esecuzione**

```
> go run floatUguali.go

Inserire un valore: 4

4 uguale a 2*2

Inserire un valore double: 10.89

10.89 uguale a 3.3*3.3
```

Provate a testare il vostro algoritmo con gli stessi valori e con valori contenenti più numeri decimali (ad es.: `7.764387267826472486`). Ottenete sempre lo stesso risultato?? Se non è così perché?

## Esercizio 4 - Dichiarazione e inizializzazione di variabili

Che cosa succede se si prova a utilizzare una variabile senza averla dichiarata? E che cosa succede se si prova a utilizzare una variabile dichiarata ma non inizializzata? Per trovare la risposta scrivete un programma che utilizzi una variabile non dichiarata e uno che stampi il valore di una variabile dichiarata ma non inizializzata. Ad esempio:

```
var a int = 3
var b int
fmt.Println("Variabile inizializzata = ", a);
fmt.Println("Variabile NON inizializzata = ", b);
fmt.Println("Variabile non dichiarata = ", c);
```

## Esercizio 5 - Analisi di frammenti di codice

Il seguente frammento di codice contiene vari errori:

```
var a, b, c int = 5, 7
if(a = b)
fmt.Println("Sono uguali")
else
fmt.Println("Sono diversi")
```

Trovate gli errori leggendo il codice e, solo successivamente, inserite il frammento di codice in un programma compilabile e verificate se le vostre congetture erano giuste.

---

Considerate il seguente frammento di codice:

```
var a int = 7
b := 5
c := a == b
```

Di che tipo deve essere la variabile `c` affinché questo codice sia compilabile? Che cosa succede quando queste istruzioni vengono eseguite avendo correttamente dichiarato la variabile `c`? Date una risposta e solo dopo inserite il frammento di codice in un programma compilabile ed eseguibile e verificate se avete o meno ragione.

---

Considerate il seguente frammento di codice:

```
var a, b, c int = 7, 5, 12
fmt.Println( a + b*c )
```

In che ordine vengono eseguite le operazioni di somma e moltiplicazione? Come è possibile invertire quest'ordine?

---

Considerate il seguente frammento di codice:

```
var zero float64 = 0
res := zero / zero
fmt.Println(res)
```

Che cosa succede quando questo frammento viene inserito in un programma? Perché? Di che tipo è la variabile `res`? Per assicurarvi di avere la risposta giusta stampate il tipo di `res` con l'istruzione

```
fmt.Printf("res = %f, tipo di res = %[1]T\n", res)
```

Notate che il codice di formattazione `%T` stampa il tipo dell'argomento a cui il codice è associato (in questo caso la variabile `res`), mentre la scrittura `[1]` dopo il simbolo di `%` dice alla funzione `Printf` che va usato il primo tra gli argomenti da stampare (ovvero `res`).

---

Considerate il seguente frammento di codice:

```
var numero float64 = 0
fmt.Println(numero)
numero = 1 / numero
fmt.Println(numero)
numero = 1 / numero
fmt.Println(numero)
numero = 1 / numero
numero = numero - numero
fmt.Println(numero)
```

Giustificate da un punto di vista matematico quello che succede quando questo frammento viene inserito in un programma ed eseguito.

## Esercizio 6 – Troncamento/Arrotondamento

Se assegnate una variabile di tipo `float64 a` una variabile di tipo `int` che cosa succede (usate l'operatore di `cast` per evitare errori)?.

Scrivere un programma che legga un numero **reale** (`float64`) e lo ***TRONCHI*** alla seconda cifra dopo la virgola.

***Suggerimento***: moltiplicate il numero per **100**, convertite il risultato in un valore intero (`int64`), poi convertitelo in `float64` e dividete il risultato per il valore **reale** `100`.

Scrivere un programma che legga un numero decimale e lo ***ARROTONDI*** alla seconda cifra dopo la virgola.

***Suggerimento:*** moltiplicate il numero per **100**, convertitelo usate il metodo `Round` del package `math` (andate a leggere l'help in linea per capire cosa permette di fare questo metodo), poi dividete il risultato per **100**.

Riscrivere i programmi precedenti generalizzandoli, in modo che il troncamento e l'arrotondamento avvengano alla **n**-esima cifra dopo la virgola, dove **n** è un valore specificato dall'utente.

## Esercizio 7 - Soluzione di equazioni di primo grado

Scrivere un programma che legga in input due valori decimali `a, b` che forniscono la descrizione di un'equazione di primo grado espressa nella forma `ax+b=0` e stampi la corrispondente radice (ovvero il valore di `x` per cui l'uguaglianza risulta verificata). Suggerimento: la soluzione è ovviamente `x=-b/a`.

**Cosa succede se a=0 e b≠0?**

**Cosa succede se a=0 e b=0?**

## Esercizio 8 - Soluzione di equazioni di secondo grado

Scrivere un programma che legga in input tre valori decimali `a, b, c` che forniscono la descrizione di un'equazione di secondo grado della forma $ax^2+bx+c=0$.

Assumete che le radici dell'equazione siano sempre reali e utilizzate il metodo `math.Sqrt(v)` del package `math` per calcolare la radice quadrata del valore contenuto nella variabile `v`.

**Utilizzando il programma appena scritto, provate a risolvere l'equazione $x^2+1=0$. Che cosa succede?**

## Esercizio 9 - Soluzione di equazioni di secondo grado – Caso generale

Per ovviare al problema emerso durante l'esercizio precedente dovete tener conto della possibilità che i radicandi siano negativi, e quindi che non sia possibile determinare delle radici reali. In questo caso si usano i numeri complessi che usano il simbolo **`i`** per indicare la radice quadrata di **`-1`** **(**$\sqrt{-1}$**)**; un esempio di numero complesso è:

`a+ib=` $a+\sqrt{-1}b$ (dove `a` **e** `b` sono numeri reali).

Scrivete la forma delle radici di un'equazione di secondo grado quando il discriminante risulta negativo e tenete conto che il simbolo **`i`** non indica il nome di una variabile ma un carattere da visualizzare appositamente. Tenuto conto di questi suggerimenti, realizzare un programma che permetta di calcolare le radici anche quando il determinante risulta negativo.

**Suggerimento**: nel caso il discriminante sia negativo, le radici dell'equazione vanno salvate in variabili di tipo `complex`.

## Esercizio 10 - Tratteggiate

Scrivete un programma che legga una riga introdotta dall'utente e la ristampi in modo che le lettere siano separate da trattini ("-").

**Esempio di esecuzione (in grassetto l'input da parte dell'utente):**

```
>go run tratteggiate.go
Inserire una frase:
```

**`Ma Ugo! NON L'ABBIAMO MAI FATTO!!`**

```
M-a U-g-o! N-O-N L'A-B-B-I-A-M-O M-A-I F-A-T-T-O!!
>go run tratteggiate.go
Inserire una frase:
```

**`OHHHH,CHE BELLO VENIRE TUTTI INSIEME A LEZIONE!!!!`**

```
O-H-H-H-H,C-H-E B-E-L-L-O V-E-N-I-R-E T-U-T-T-I I-N-S-I-E-M-E A L-E-Z-I-O-N-E!!!!
```

## Esercizio 11 - Palindrome

Una parola è ***palindroma*** se è uguale quando viene letta da destra a sinistra e da sinistra a destra. Quindi "enne" è palindroma, ma "papa" non lo è. Scrivete un programma che legga una stringa da tastiera e dica se è palindroma o no.

**Esempio di esecuzione (in grassetto l'input da parte dell'utente):**

```
>go run palindrome.go
```

**abba**

```
abba è palindroma
>go run palindrome.go
```

**pippo**

```
pippo non è palindroma
```

## Esercizio 12 - Lunghezza media delle parole

Scrivete un programma che legga una un testo da file e calcoli quante parole contiene e la loro lunghezza media.

**Esempio di funzionamento**

Se il file mioTesto.txt contenesse il seguente testo:

```
S'i' fosse foco,
arderei 'l mondo
```

lanciando il programma con il comando di **redirezione**[1] da input si otterrebbe:

```
>go run lungMediaParole.go < mioTesto.txt
Inserisci un testo:
Il testo contiene 7 parole, con lunghezza media 3.43.
```

**Suggerimenti**

[1] Ricordiamo che il comando di redirezione rappresentato dal simbolo di minore "<" dice che l'input da standard input viene rediretto da un file, o viene viene preso da un file. in questo caso il file è "`mioTesto.txt`"

Una parola è una sequenza di caratteri alfabetici delimitata da spazi o caratteri non alfabetici (e.g. ! , ; ). Per distinguere le parole potete controllare quando comincia e quando finisce una sequenza di caratteri alfabetici.

Per vedere se un carattere è alfabetico potete usare la funzione `unicode.IsLetter(r)` del package `unicode`.

Se `s` è una stringa, il seguente frammento di codice conta i caratteri alfabetici e in caratteri non alfabetici in `s`:

```
var count int = 0
var s string = "I miei caratteri >!!alfabetici!!< non sono: , ; ! % $ / ()"
for _, r := range s {
    if unicode.IsLetter(r) {
        count++
    }
}
fmt.Println("Stringa:", s, "\nCaratteri alfabetici = ", count, "\nCaratteri NON alfabetici =", ???)
```

**Nel frammento di codice sopra ci sono dei punti di domanda: avendo contato i caratteri alfabetici in `s` provate voi a stabilire come determinare quelli non alfabetici!!**

## Esercizio 13 - La prova del nove

La prova del nove è un meccanismo per il controllo dei calcoli basato sull'aritmetica modulare.

Essa sfrutta il fatto che:

$$x \cdot y = z \Rightarrow (x \cdot y) \bmod 9 = z \bmod 9,$$

dove mod è il resto della divisione intera (`%`).

Fate però attenzione: **il viceversa non è sempre vero**. Altrimenti detto:

$(x \cdot y) \bmod 9 = z \bmod 9$ **NON VALE NECESSARIAMENTE CHE** $x \cdot y = z$

In termini matematici:

$x \cdot y = z \Rightarrow (x \cdot y) \bmod 9 = z \bmod 9$ **MA**

$(x \cdot y) \bmod 9 = z \bmod 9 \Rightarrow (x \cdot y = z \vee x \cdot y \neq z)$

Scrivete un programma che scorra tutte le terne di interi positivi `x, y, z` minori di un intero inserito dall'utente da standard input e che stampi quelle per cui non vale il viceversa cioè:

$$(x \cdot y) \bmod 9 = z \bmod 9 \quad \textbf{MA} \quad x \cdot y \neq z$$

**Esempio di esecuzione (in grassetto l'input da parte dell'utente):**

```
>go run provaDelNove.go
Inserisci soglia: 5
3,  3,  0
3,  4,  3
4,  3,  3
>go run provaDelNove.go
Inserisci soglia: 7
2,  5,  1
2,  6,  3
3,  3,  0
3,  4,  3
3,  5,  6
3,  6,  0
4,  3,  3
4,  5,  2
4,  6,  6
5,  2,  1
5,  3,  6
5,  4,  2
5,  6,  3
6,  2,  3
6,  3,  0
```

```
6,  4,  6
6,  5,  3
6,  6,  0
```

## Esercizio 14 - Figure

Scrivete 2 programmi che prendono in input un intero `n` e producono in output una figura di `n` righe e `n` colonne, costituita da simboli * (asterisco) e + (più) non intervallati da spazi. La differenza fra un programma e l'altro è il modo in cui sono alternati i simboli, come indicato negli esempi che seguono.

**Primo programma:**

In questo caso, il programma disegna solo figure di lato `n` pari. Per tal motivo, se l'utente inserisce un valore dispari per `n`, il programma lo trasforma in un numero pari (sommando 1 al numero) e lo notifica all'utente con una stampa di tipo "`numero inserito dispari: sommo 1`".

**Esempi di funzionamento (in grassetto il valore inserito dall'utente):**

```
>go run figure4.go
Inserisci un numero: 7
Numero dispari: sommo 1
+++*+++
++*+*++
+*+++*+
*+++++*
*+++++*
+*+++*+
++*+*++
+++*+++
>go run figure4.go
Inserisci un numero: 10
++++*++++
+++*++*+++
++*++++*++
+*++++++*+
*++++++++*
*++++++++*
```

```
+*******+
++*****++
+++***+++
++++*++++
```

**SUGGERIMENTO**

Considerate la seguente rappresentazione della stampa del programma per `n` = 6:

| i \ j | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | + | + | * | * | + | + |
| 1 | + | * | + | + | * | + |
| 2 | * | + | + | + | + | * |
| 3 | * | + | + | + | + | * |
| 4 | + | * | + | + | * | + |
| 5 | + | + | * | * | + | + |

Per ogni riga, le occorrenze degli asterischi si hanno per queste coppie di valori assegnati agli indici i e j:

| Riga i-esima | (i,j) – Prima occorrenza di * | (i,j) – Seconda occorrenza di * |
|---|---|---|
| 0 | (0,2) | (0,3) |
| 1 | (1,1) | (1,4) |
| 2 | (2,0) | (2,5) |
| 3 | (3,0) | (3,5) |
| 4 | (4,1) | (4,4) |
| 5 | (5,2) | (5,3) |

Notate qualche proprietà soddisfatta dalle coppie di valori riportate in tabella? Quindi ...

**Secondo programma:**

Questo programma è uguale al precedente ma disegna solo figure di lato `n` dispari. Per tal motivo, se l'utente inserisce un valore pari per `n`, il programma lo trasforma in un numero dispari (sommando 1 al numero) e lo notifica all'utente con una stampa di tipo "`numero inserito pari: sommo 1`".

**Esempi di funzionamento (in grassetto il valore inserito dall'utente):**

```
>go run figure4.go
Inserisci un numero: 7
+++*+++
++*+*++
+*+++*+
*++++*
+*+++*+
++*+*++
+++*+++
>go run figure4.go
Inserisci un numero: 10
Numero pari: sommo 1
++++*++++
+++*++*+++
++*+++*++
+*++++*+
*++++++*
+*++++*+
++*++++*++
+++*++*+++
++++*++++
++++*++++
```