# Laboratorio 2

**Elementi di Informatica e Programmazione**

## Esercizio 1 - Confronti tra stringhe

Progettare il programma `sortThreeStrings.py` che

- chieda all'utente di inserire tre stringhe (una per riga)
- visualizzi le stringhe in ordine lessicografico crescente (una per riga)

Il programma deve terminare con queste quattro righe di codice:

```
print("Le stringhe in ordine crescente sono:")
print(min_str)
print(mid_str)
print(max_str)
```

E non deve avere altri enunciati di `print`. In altre parole dovete progettare un programma che assegni alle variabili `min_str`, `mid_str` e `max_str` rispettivamente la prima, la seconda e la terza stringa in ordine lessicografico, tra quelle date.

**ATTENZIONE**: ovviamente una soluzione è inserire le tre stringhe in una lista, usare il metodo `list.sort()` per ordinarla e leggere le stringhe ordinate nella lista. L'obiettivo di questo esercizio è tuttavia di ragionare sull'ordine lessicografico e sui confronti, per cui provate a risolverlo usando solo statement condizionali (`if`, `elif` e `else`).

Alcuni suggerimenti:

- Ipotizzando che nella valutazione delle espressioni booleane l'interprete Python NON utilizzi la tecnica del "cortocircuito", calcolare il numero massimo di confronti impiegati dall'algoritmo implementato nella soluzione, nel caso peggiore. Esiste una soluzione che richiede, al massimo, **TRE soli confronti**: cercare di individuarla e realizzarla.

- *Nota*: "una soluzione che richiede, al massimo, tre confronti" significa "una soluzione che, nel momento in cui viene eseguita, effettua al massimo tre confronti, indipendentemente dal valore dei dati in esame". NON significa che ci debbano essere soltanto tre enunciati if… significa che devono essere eseguiti soltanto tre confronti. Ad esempio, l'esecuzione di questo frammento di codice richiede al massimo DUE confronti (soltanto uno quando x >= y).

```
if x < y:
    if z < y:
        print("pippo")
    else:
        print("pluto")
else:
    print("topolino")
```

- Per collaudare il programma, utilizzare tre stringhe molto semplici, ad esempio A, B e C. Eseguire ripetutamente il programma fornendo in ingresso tutte le (6) possibili permutazioni di tali stringhe, verificando che il programma le stampi sempre nell'ordine corretto, che è (ovviamente) A, B, C, indipendentemente dall'ordine in cui vengono fornite in input.
- Esperimento da condurre: dove si colloca la stringa vuota nell'ordinamento lessicografico? Precede o segue qualsiasi stringa?

### Soluzione

In una prima versione semplicemente elenchiamo i 6 casi possibili. Questa versione tuttavia fa ben più di tre confronti, nel caso peggiore.

```
# sortThreeStrings.py
print("Inserire tre stringhe, una per riga")
s1 = input()
s2 = input()
s3 = input()
if s1 <= s2 <= s3 :
    min = s1
    mid = s2
    max = s3
elif s1 <= s3 <= s2 :
    min = s1
    mid = s3
    max = s2
elif s2 <= s1 <= s3 :
```

```
    min = s2
    mid = s1
    max = s3
elif s2 <= s3 <= s1 :
    min = s2
    mid = s3
    max = s1
elif s3 <= s1 <= s2 :
    min = s3
    mid = s1
    max = s2
else : # s3 <= s2 <= s1
    min = s3
    mid = s2
    max = s1
print("Le stringhe in ordine crescente sono:")
print(min)
print(mid)
print(max)
```

Questa seconda versione risolve il problema facendo al massimo tre confronti.

```
# sortThreeStrings2.py
print("Inserire tre stringhe, una per riga")
s1 = input()
s2 = input()
s3 = input()
if s1 <= s2 :
    if s1 <= s3 :
        min = s1
        if s2 <= s3 :
            mid = s2
            max = s3
        else :
            mid = s3
            max = s2
    else :
        min = s3
        mid = s1
        max = s2
else : # s2 < s1
    if s2 <= s3 :
```

```
        min = s2
        if s1 <= s3 :
            mid = s1
            max = s3
        else :
            mid = s3
            max = s1
    else :
        min = s3
        mid = s2
        max = s1
print("Le stringhe in ordine crescente sono:")
print(min)
print(mid)
print(max)
```

## Esercizio 2 - Colonie di batteri

Un biologo ha bisogno di un programma `batteri.py` per prevedere la crescita di una popolazione di batteri in un certo intervallo di tempo di osservazione. Gli input sono:

- il numero iniziale di organismi
- il tasso di crescita all'ora in percentuale
- l'intervallo (in ore) di osservazione

Supponendo che nessun organismo muoia si scriva un programma che accetti questi input e visualizzi una previsione del numero totale di batteri alla fine dell'intervallo di osservazione. Esempio di esecuzione

- numero iniziale di organismi: 5
- tasso di crescita all'ora in %: 2
- intervallo (in ore) di osservazione: 24

*Previsione*: il numero di batteri dopo 24.0 ore è uguale a 5.

*Suggerimento*: abbiamo visto qualcosa di *estremamente* simile in classe, nella nostra introduzione ai cicli.

### Soluzione

```
num_organisms = int(input("numero iniziale di organismi: "))
growth_rate = float(input("tasso di crescita all'ora in %: "))
time_interval = int(input("intervallo (in ore) di osservazione: "))

time = 0
while (time < time_interval):
    time = time + 1
    num_organisms = num_organisms + round(num_organisms*growth_rate/100)

print("il numero di batteri dopo", time_interval, "ore è uguale a", num_organisms)
```

## Esercizio 3a - da binario a decimale

Scrivere un programma `binarytodecimal.py` che riceva in input un numero binario, lo converta in decimale e lo stampi in output

### Soluzione

Questa soluzione implementa l'algoritmo basato sulla notazione polinomiale dei numeri visto a lezione. In particolare, la somma viene fatta a partire dalla cifra più significativa, ovvero il termine di grado più alto nel polinomio.

```
bstring = input("Enter a string of bits: ")
decimal = 0
exponent = len(bstring) - 1
for digit in bstring:
    assert digit == "0" or digit == "1", "Il numero in input deve contenere solo cifre bin
    decimal = decimal + int(digit) * 2 ** exponent
    exponent = exponent - 1
print("The integer value is", decimal)
```

## Esercizio 3b - da decimale a binario

Scrivere un programma `decimaltobinary.py` che riceva in input un numero decimale, lo converta in binario e lo stampi in output

### Soluzione

Il codice seguente implementa l'algoritmo visto a lezione per la conversione da decimale a binario, procedendo per divisioni successive.

```
decimal = int(input("Enter a decimal integer: "))
if decimal == 0:
    print(0)
else:
    print("Quotient Remainder Binary")
    bstring = ""
    while decimal > 0:
        remainder = decimal % 2
        decimal = decimal // 2
        bstring = str(remainder) + bstring
    print("The binary representation is", bstring)
```

## Esercizio 4 - Statistiche su numeri

Progettare il programma `someNumbers.py` che data una lista di numeri

chieda all'utente di inserire una sequenza di numeri (terminata dalla parola chiave `"fine"`) e ne visualizzi:

- la somma
- la somma dei valori assoluti
- il prodotto
- il valore medio

Il programma deve visualizzare un messaggio d'errore (e niente altro) se la sequenza di numeri termina prima che siano stati introdotti almeno due numeri.

### Soluzione

```
count = 0   # serve per fare la media e per segnalare errore se < 2
sum = 0
sumOfAbs = 0
product = 1 # attenzione all'inizializzazione del prodotto... non zero!
print("Inserisci un numero per riga terminando con la parola `fine`")
while True :
    s = input()
    if s.strip().lower() == "fine" :
```

```
        # usciamo quando l'utente digita `fine`. Usiamo i metodi
        # strip e lower per essere robusti rispetto a eventuali spazi
        # bianchi inseriti e all'utilizzo di maiuscole/minuscole
        break
    n = int(s)
    sum += n
    sumOfAbs += abs(n)
    product *= n
    count += 1
if count < 2 :
    print("Servono almeno due numeri")
else :
    print("Somma:", sum)
    print("Somma dei valori assoluti:", sumOfAbs)
    print("Prodotto:", product)
    print("Valore medio:", (sum / count)) # le parentesi interne non sono necessarie
```

## Esercizio 5 - Massimo Comun Divisore

Scrivere il programma `euclideMCD.py` che calcoli e visualizzi il massimo comun divisore (MCD) di due numeri interi positivi `m` e `n` forniti dall'utente (con una segnalazione d'errore se l'utente fornisce numeri interi non positivi). Si usi il ben noto Algoritmo di Euclide per il calcolo del MCD di due numeri interi positivi `m` ed `n` (con `m` $>$ `n`):

1. Finchè il resto della divisione di `m` per `n` è diverso da zero
   - il nuovo valore di `m` è uguale al precedente valore di `n`
   - il nuovo valore di n è uguale al resto della divisione del precedente valore di `m` per il precedente valore di `n`
2. Il MCD è l'attuale valore di `n`

### Soluzione

```
m = int(input("Primo numero (intero positivo)? "))
n = int(input("Secondo numero (intero positivo)? "))

# Uso assert per verificare che i numeri siano positivi
assert m > 0 and n > 0, "Non sono due numeri positivi"

origM = m # servono per poi visualizzare il risultato
```

```
origN = n # perché m e n vengono modificati
# Faccio in modo che sia m >= n
if m < n :
    # scambio m con n usando una variabile temporanea
    tmp = m
    m = n
    n = tmp
# ora certamente m >= n
# Algoritmo di Euclide
while m % n != 0 :
    tmp = n
    n = m % n
    m = tmp
# ora n è il MCD di origM e origN
print("MCD(" + str(origM) + "," + str(origN) + ") = " + str(n))
```

## Esercizio 6 - Mesi dell'anno

Progettare il programma `printSelectedMonth.py` che chieda all'utente un numero intero compreso tra 1 e 12 e visualizzi il nome del mese corrispondente, come in questo esempio di funzionamento:

```
> Inserisci il numero del mese (1-12): 5
Maggio
```

### Soluzione

```
# Definisco una lista con i nomi dei mesi
months = [
    "Gennaio",
    "Febbraio",
    "Marzo",
    "Aprile",
    "Maggio",
    "Giugno",
    "Luglio",
    "Agosto",
    "Settembre",
    "Ottobre",
    "Novembre",
```

```
    "Dicembre"
]

# Chiedo il numero in input all'utente
input_month = int(input("Inserisci il numero del mese (1-12): "))
# Verifico che il numero sia tra 1 e 12
assert 1 <= input_month <= 12

print(months[input_month-1])
```

**Importante**: in questo esercizio non si può usare l'enunciato `if`!

### Esercizio 7 - Numeri primi

Progettare il programma `printPrimes.py` che calcoli e visualizzi in ordine crescente, un numero per riga, tutti i numeri primi fino a un valore massimo introdotto inizialmente dall'utente; chiedere ripetutamente tale valore all'utente finché non viene introdotto un numero intero maggiore di uno. Ricordare che un numero intero maggiore di uno è primo se è divisibile con resto nullo soltanto per il numero uno e per se stesso.

#### Soluzione

```
while True :
    maxValue = int(input("Inserisci il valore massimo, intero maggiore di uno: "))
    if maxValue > 1 :
        break
print("Numeri primi non maggiori di", maxValue, ":")
i = 2
while i <= maxValue :
    isPrime = True
    # Domanda:
    # nel ciclo sottostante, è veramente necessario
    # imporre la condizione j < i, oppure si potrebbe
    # usare un vincolo più restrittivo
    # (cioè j < di una quantità minore di i) ???
    j = 2
    while j < i and isPrime :
        if i % j == 0 :
            isPrime = False
        j += 1
```

```
    if isPrime :
        print(i)
    i += 1
```

## Esercizio 8 - Sottostringhe

Scrivere il programma `isSubstring.py` che

- chieda all'utente di introdurre due stringhe (una per riga), `s1` e `s2`; ciascuna stringa è costituita da tutti i caratteri presenti sulla riga, compresi eventuali spazi iniziali, finali e/o intermedi
- verifichi (visualizzando al termine un messaggio opportuno) se la seconda stringa, `s2`, è una sottostringa di `s1`, cioè se esiste una coppia di numeri interi non negativi, x e y, per cui `s1`[x:y] restituisce una stringa uguale a `s2`

**Il programma non deve utilizzare metodi che operano su stringhe.**

Verificare che il programma gestisca correttamente la situazione in cui `s2` è la stringa vuota, che deve risultare sottostringa di qualsiasi stringa. Come è invece ragionevole che si comporti il programma se `s1` è la stringa vuota?

### Soluzione

```
s1 = input("Prima stringa: ")
s2 = input("Seconda stringa: ")
foundSubstring = False
# l'inizializzazione precedente significa
# "non ho ancora trovato la sottostringa", quindi è corretta
i = 0
while i < len(s1) - len(s2) + 1 and not foundSubstring :
    # cerchiamo s2 in s1 a partire dal carattere i-esimo di s1
    #
    # appena troviamo una corrispondenza, questo ciclo termina
    # (osservate la condizione di terminazione, con l'operatore and)
    #
    # osservate il valore massimo assunto da i: non ha senso cercare
    # in s1 a partire da una posizione in cui il numero di caratteri
    # successivi è minore del numero di caratteri presenti in s2 !!
    if s1[i : i + len(s2)] == s2 :
        foundSubstring = True
    i += 1
```

```
print("La seconda stringa", end="")
if not foundSubstring :
    print(" non", end="")
print(" è una sottostringa della prima")
```

- Cosa succede se s2 è la stringa vuota?
- Il programma dovrebbe segnalare che s2 è una sottostringa, indipendentemente dal contenuto di s1… Lo fa? Perché?
- Cosa succede se s1 è la stringa vuota? Il programma dovrebbe segnalare che s2 è una sottostringa soltanto se anche s2 è vuota… Lo fa? Perché?

## Esercizio 9 - Il gioco di Nim

Si tratta di un gioco con un certo numero di varianti: utilizzeremo la versione seguente, che ha una strategia interessante per arrivare alla vittoria. Due giocatori prelevano a turno biglie da un mucchio. In ciascun turno, il giocatore sceglie quante biglie prendere: deve prenderne almeno una, ma non oltre la metà del mucchio. Perde chi è costretto a prendere l'ultima biglia.

Scrivere il programma `nim.py` con cui un giocatore umano possa giocare a Nim contro il computer. Si genera un numero intero casuale, compreso fra 10 e 100, per indicare il numero iniziale di biglie. Si decide casualmente se la prima mossa tocca al computer o al giocatore umano. Si decide casualmente se il computer giocherà in modo intelligente o stupido. Nel modo stupido, quando è il suo turno, il computer si limita a sottrarre dal mucchio un numero casuale di biglie, purché sia una quantità ammessa. Nel modo intelligente, il computer preleva il numero di biglie sufficiente affinché il numero di quelle rimanenti sia uguale a una potenza di due diminuita di un'unità, ovvero 1, 3, 7, 15, 31 o 63: ricordate però che la mossa è valida se il giocatore non prende più della metà delle biglie del mucchio. Si può dimostrare che, se il computer preleva il numero di biglie sufficiente affinché il numero di quelle rimanenti sia uguale a una potenza di due diminuita di un'unità (biglie rimanenti = 1, 3, 7, 15, 31 o 63), si tratta sempre di una mossa valida, eccetto quando il numero delle biglie è uguale a una potenza di due diminuita di un'unità. In caso di mossa non valida il computer preleverà una quantità di biglie casuale, purché ammessa

Si noti che, nel modo intelligente, il computer non può essere sconfitto quando ha la prima mossa, a meno che il mucchio non contenga inizialmente 15, 31 o 63 biglie. Nelle medesime condizioni, un giocatore umano che abbia la prima mossa e che conosca tale strategia vincente, può ovviamente vincere contro il computer.

Il programma deve giocare una partita dopo l'altra, fino a quando l'utente non introduce un opportuno comando che interrompe il gioco. Ogni partita è completamente indipendente dalle

precedenti, sia per numero di biglie iniziali, sia per stupidità/intelligenza del computer, sia per assegnazione della prima mossa.

Per risolvere un problema articolato come questo, occorre procedere "per gradi", individuando soluzioni di problemi intermedi che si possano collaudare, in modo da aggiungere funzionalità a uno schema già funzionante, altrimenti, se si collauda il programma soltanto alla fine, diventa **estremamente** difficile diagnosticare i problemi che si manifestano (e che ci saranno senz'altro…).

Un possibile schema da seguire potrebbe essere questo:

1. impostare un ciclo (virtualmente) infinito, il cui corpo conterrà tutto ciò che serve per giocare una partita, ma che per il momento abbia soltanto il compito di chiedere all'utente se vuole giocare un'altra partita, gestendo accuratamente tutti i casi ("Vuoi giocare ancora?": risposta S, risposta N, risposta diversa); questa fase di input richiede, a sua volta, un ciclo, che termini soltanto quando l'utente risponde S o N; risolto il problema, collaudare il programma in tutti i casi possibili e verificare che il comportamento sia sempre corretto
2. arricchire il corpo del ciclo principale (quello impostato al passo precedente), aggiungendo la fase di inizializzazione dei parametri di gioco (chi giocherà per primo? il computer sarà "intelligente" oppure no? quante biglie saranno inizialmente in gioco? Ricordate che potete usare il modulo `random` di Python per generare numeri casuali), con temporanea visualizzazione di tali parametri, procedendo a un loro collaudo con più esecuzioni, verificando anche che i parametri cambino di partita in partita se il giocatore chiede di giocare di nuovo (senza eseguire di nuovo il programma)
3. concludere la soluzione del problema inserendo nel corpo del ciclo principale, tra la fase di inizializzazione dei parametri e la fase di richiesta "Vuoi giocare ancora?", un ciclo che gestisca i turni (alternando i due giocatori, umano e computer), tenendo conto della quantità di biglie rimaste nel mucchio e decidendo quando la partita è finita, decretando il vincitore; collaudare il programma completo mediante molteplici esecuzioni

### Soluzione

Inutile dire che un programma così lungo e articolato si può risolvere in molti modi diversi… questa è solo una delle possibili soluzioni (come sempre, ma in questo caso in modo particolare). Per questo motivo, se la vostra soluzione dovesse essere diversa da quella proposta non preoccupatevi!

```
from random import randint, random
while True : # terminerà (ovviamente) con un break
    # inizio di una (nuova) partita
    numBalls = randint(10, 100)
    # le prossime 4 righe di codice attribuiscono un valore casuale
```

```
# a una variabile booleana... i due valori possibili (True e False)
# sono equiprobabili, perché uso come soglia il valore intermedio
# dell'intervallo dei numeri generati (che va da 0 a 1);
# usando come soglia, ad esempio, 0.3, otterrei il valore False
# nel 30% dei casi (e, ovviamente, il valore True nel 70%), cioè
# i due valori non sarebbero equiprobabili (ma a volte può essere
# utile anche tale situazione)
if random() < 0.5 :
    computerPlayerIsExpert = False
else :
    computerPlayerIsExpert = True
# le prossime 4 righe di codice attribuiscono un valore casuale
# a una variabile booleana...
if random() < 0.5 :
    nextPlayerIsHuman = False
else :
    nextPlayerIsHuman = True
while numBalls > 1 :
    if nextPlayerIsHuman : # è il turno del giocatore umano
        repeat = True
        while repeat :
            print("Biglie presenti nel mucchio: ", numBalls)
            taken = int(input("Quante ne vuoi prendere? "))
            if taken < 1 or taken > (numBalls // 2) : # attenzione che sia
                                                     # corretto anche se
                                                     # dispari...
                print("Azione errata") # repeat rimane True
            else :
                numBalls -= taken
                repeat = False # il giocatore umano ha fatto una scelta valida
    else : # è il turno del computer
        if not computerPlayerIsExpert : # computer "stupido"
            # deve prendere almeno una biglia e un numero di biglie
            # che non superi la metà di quelle presenti...
            # verificare che la scelta sia valida sia in caso di numero
            # di biglie pari sia in caso di numero di biglie dispari
            taken = randint(1, numBalls // 2)
        else : # computer "intelligente"
            if (numBalls == 3 or numBalls == 7 or numBalls == 15
                              or numBalls == 31 or numBalls == 63) :
                # temporaneamente stupido... deve scegliere a caso
```

```
                taken = randint(1, numBalls // 2)
            elif numBalls > 63 : # attenzione all'ordine in cui faccio
                                 # i confronti...
                taken = numBalls - 63
            elif numBalls > 31 :
                taken = numBalls - 31
            elif numBalls > 15 :
                taken = numBalls - 15
            elif numBalls > 7 :
                taken = numBalls - 7
            elif numBalls > 3 :
                taken = numBalls - 3
            else : # ci sono certamente 2 biglie
                taken = 1
        print("Biglie presenti nel mucchio: ", numBalls)
        print("Il computer ne ha prese: ", taken)
        numBalls -= taken
    # cosa fa l'enunciato seguente? si usa spesso per cambiare il
    # valore di una variabile booleana
    nextPlayerIsHuman = not nextPlayerIsHuman
# a questo punto numBalls vale certamente 1
# quindi chi deve fare la prossima mossa ha perso
if nextPlayerIsHuman :
    print("Hai perso")
else :
    print("Hai vinto!!")
if computerPlayerIsExpert :
    print('Il computer giocava in modo "intelligente"')
else :
    print('Il computer giocava in modo "stupido"')
while True :
    response = input("Vuoi giocare ancora? (S/N) ")
    if response.upper() == "N" :
        playAgain = False
        break
    elif response.upper() == "S" :
        playAgain = True
        break
if not playAgain :
    break
```