Esercizi su Python

14 maggio 2012

1. Scrivere una procedura che generi una stringa di 100 interi casuali tra 1 e 1000 e che:
   1. conti quanti elementi pari sono presenti nella lista;
   2. conti quanti quadrati perfetti sono presenti nella lista.
2. Vale che

$$\log(1+x) \;=\; \sum_{n=1}^{\infty}(-1)^{n+1}\,\frac{x^n}{n} \quad \text{per} \quad -1 < x \leq 1$$

   Scrivere una procedura che determini il primo numero intero N tale per cui valga (avendo posto $x = 1$)

$$\left|\left(\sum_{n=1}^{N}(-1)^{n+1}\,\frac{1^n}{n}\right) - \log(2)\right| \;\leq\; 2\cdot 10^{-4}$$

   Per utilizzare le funzioni matematiche *logaritmo naturale* e *valore assoluto* è necessario inserire all'inizio della procedura la riga di comando

```
import math
```

   A questo punto le due funzioni si possono richiamare con

```
math.log(argomento)
math.fabs(argomento)
```

3. Scrivere una procedura che determini tutti i numeri primi compresi tra 1 a 100 utilizzando il Crivello di Eratostene, ovvero generare una lista i cui elementi siano i numeri da 1 a 100, da cui si elimina 1 ed in seguito si eliminano di volta in volta i multipli di 2, poi i multipli di 3, i multipli

di 5, ecc.
Utilizzare questa lista per verificare la Congettura di Goldbach, la quale afferma che ogni numero pari maggiore di 2 si può scrivere come somma di due numeri primi (anche uguali), per i numeri fino a 100.
Stampare a schermo tutte le possibili scritture di tali numeri come somma di due primi (considerando anche scritture "simmetriche", come $8 = 3 + 5$ e $8 = 5 + 3$).

La procedura per il Crivello di Eratostene può essere scritta nel modo seguente:

1. costruire una lista con i numeri da 1 a 100;
2. assegnare 0 all'elemento di posto 0 (che prima aveva valore uguale ad uno);
3. far partire un ciclo di iterazione attraverso il quale ogniqualvolta si incontra in posizione `i` un valore diverso da zero, si assegna zero a tutti gli elementi di posizione data dai multipli di `i` meno 1 (ovvero, alla prima iterazione il primo valore non nullo che si incontra è 2, che si presenta in posizione 1, allora si assegna zero a tutti gli elementi in posizione $3, 5, 7, \ldots, 99$; poi si avanza nell'iterazione: il primo valore non nullo seguente è il 3, che si trova in posizione 2, allora si assegna zero a tutti gli elementi in posizione $5, 8, 11, \ldots$ e così via);
4. alla fine dell'iterazione si ottiene una lista formata solamente da zeri e da numeri primi, e da essa si crea una nuova lista formata solo dagli elementi non nulli della precedente.

4. Scrivere una procedura che effettui la divisione tra due polinomi in una variabile a coefficienti reali attraverso l'algoritmo di riduzione.
Per fare questo si implementa il seguente algoritmo:

```
 INPUT:   f, g polinomi con g non nullo
 OUTPUT:  q, r polinomi con f = q * g + r
          ed r = 0 oppure d(r) < d(f)
ALGORITMO:
      r = f
      q = 0
      FINO A CHE (r != 0  E  d(r) > d(f)) RIPETI:
            q = q + (lc(r)/lc(g)) * x^(d(r) - d(g))
            r = r - (lc(r)/lc(g)) * x^(d(r) - d(g)) * g
      RESTITUISCI q, r
```

dove:
`d(f)` = grado di f quando $f \neq 0$
`lc(f)` = coefficiente direttivo di f (coefficiente del monomio di grado più alto)
`lm(f)` = monomio direttivo di f (monomio di grado più alto)

L'implementazione della procedura può avvenire in questo modo: si pensano i polinomi come delle liste, date dai loro coefficienti, in cui l'elemento di posizione più alta corrisponde al coefficiente direttivo del polinomio. A questo punto la procedura può essere composta dalle parti seguenti:

- una funzione che elimini gli eventuali zeri "in coda" al polinomio che si possono presentare nel processo di riduzione, ovvero, se ad essa viene data come input la lista $[2, 0, 3, 0, 0]$, essa deve restituire la lista $[2, 0, 3]$;
- una funzione che prenda in input una lista e verifichi se essa è la lista vuota, che rappresenta il polinomio nullo (questa funzione lavora correttamente solo se la lista in input non presenta zeri "in coda");
- una funzione che effettui la divisione vera e propria tramite riduzioni successive (implementate attraverso un ciclo di iterazioni), che verrà richiamata dopo aver controllato che il polinomio g non è il polinomio nullo.

5. Scrivere una procedura che, data una lista di elementi diversi, restituisca una lista di liste contenenti tutte le permutazioni possibili degli elementi assegnati. Ovvero, se il programma riceve $[1, 2, 3]$, allora esso dovrà restituire

$$\big[\,[1,2,3],[3,1,2],[2,3,1],[3,2,1],[1,3,2],[2,1,3]\,\big]$$

(Questo problema si può risolvere in modo ricorsivo.)

6. Scrivere una procedura che generi una lista di 100 numeri interi casuali da 1 a 1000 e che prenda in input un numero intero.
Per ottenere dei numeri interi casuali è possibile utilizzare la funzione `randint` del modulo `random`. Per fare ciò, all'inizio della procedura è necessario inserire la riga di comando:

```
import random
```

La funzione può dunque essere richiamata nel modo seguente:

`random.randint(1,1000)`

A questo punto la procedura deve ordinare in modo crescente gli elementi della lista secondo l'algoritmo seguente (detto *bubble sort*):

> Ad ogni iterazione, l'algoritmo confronta il primo elemento con il secondo e se necessario li scambia affinché siano in ordine, poi esegue il confronto tra il secondo ed il terzo, e così via. In questo modo, dopo la prima iterazione, l'elemento massimo della lista si trova all'ultimo posto. Si esegue dunque questa iterazione per $n-1$ volte, dove n è la lunghezza della stringa, e si ottiene una stringa ordinata.

Si rimuovono ora gli eventuali elementi ripetuti (che dopo l'ordinamento si troveranno in posizioni adiacenti).
Infine si controlla se il numero intero preso in input sia o meno presente all'interno della lista tramite un algoritmo di *ricerca binaria*:

> Si confronta il numero in input con il valore dell'elemento della lista di posizione $\lfloor \frac{n}{2} \rfloor$ (parte intera inferiore di $\frac{n}{2}$), dove n è la lunghezza della stringa, ed a seconda dell'esito del confronto si possono avere tre situazioni:
>
> i. se i due numeri confrontati sono uguali, l'algoritmo termina con esito positivo;
> ii. se il numero in input è maggiore del valore dell'elemento di mezzo, l'algoritmo richiama se stesso con argomento non più tutta la lista, ma solo la seconda metà della stessa;
> iii. se il numero in input è minore del valore dell'elemento di mezzo, l'algoritmo richiama se stesso con argomento non più tutta la lista, ma solo la prima metà della stessa;
>
> Quando si giunge ad una lista con un solo elemento l'algoritmo termina con esito positivo o negativo.

7. Considerare la seguente situazione: nell'angolo in alto a sinistra di una scacchiera 8×8 completamente libera viene posto un Cavallo; esso è libero di muoversi nel modo usuale negli scacchi (ovvero ad "L", spostandandosi di due caselle in alto/basso e di una a sinistra/destra, oppure di una casella a sinistra/destra e di due in alto/basso) rimanendo all'interno della scacchiera. Ci si chiede se esista una sequenza di mosse tale per cui, partendo dall'angolo in alto a sinistra, il Cavallo riesca a toccare tutte e 64 le caselle della scacchiera senza mai passare due volte per la stessa casella.

Scrivere una procedura che aiuti ad ottenere informazioni sulla questione precedente nel seguente modo: si simula l'azione di un giocatore che compie le proprie mosse in modo casuale, ma sempre in modo "legale" (ovvero non uscendo mai dalla scacchiera) e non ritornando mai su caselle che ha già toccato, e che si ferma quando non è più possibile effettuare mosse consentite. Per fare questo, realizzare la scacchiera sotto forma di matrice (ovvero come una lista annidata) e pensare alla posizioni come a liste di due elementi, dati dagli indici dell'elemento della matrice che corrisponde alla posizione considerata. Inizialmente la matrice sarà composta da soli zeri, e mano a mano che il Cavallo si muoverà nelle caselle toccate dalle mosse comparirà un numero che indicherà quante mosse sono state compiute fino ad ora.
Ad esempio, quindi, dopo 3 mosse (compresa quella iniziale, che pone il Cavallo nella posizione in alto a sinistra, ovvero $[0, 0]$) uno dei possibili scenari potrebbe essere questo:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

In particolare, la procedura può prevedere:

- Una funzione `mossa` che esegua una mossa "legale" del Cavallo, ovvero essa prende in input una posizione `posizione` ed un numero intero `numero` da 0 a 7 e restituisce in output la posizione ottenuta applicando la mossa `numero` alla posizione `posizione` (se ciò non è possibile perché tale mossa farebbe uscire il Cavallo dalla scacchiera, la funzione restituisce `posizione`). Si nota come questa funzione non tenga conto della regola per cui il Cavallo non deve passare due volte sulla stessa casella.
- Una funzione `mosse_possibili` che prende in input una posizione `posizione` e la matrice `matrice` della configurazione corrente della scacchiera e restituisce una lista delle mosse possibili che il Cavallo può compiere partendo da `posizione`. Essa dunque richiama la funzione `mossa` per tutte le otto mosse possibili, e restituisce soltato le mosse legali e quelle che non fanno arrivare il Cavallo su una delle posizioni che ha precedentemente toccato.

- Una funzione `gioco` che inizializza una matrice 8×8 `matrice` composta da tutti zeri ed imposta la posizione corrente `current` a $[0, 0]$. A questo punto parte un ciclo di 64 iterazioni, in ciascuna delle quali si svolgono le seguenti operazioni:

  Si richiama `mosse_possibili` con argomenti lo posizione corrente e la matrice della configurazione corrente della scachiera. A questo punto vi sono due possibilità: se la lista restituita da `mosse_possibili` è vuota allora il gioco non può più proseguire, dunque il ciclo di iterazione deve terminare, restituendo un messaggio che indichi dopo quante mosse il gioco si è fermato; se la lista restituita da `mosse_possibili` non è vuota, allora si assegna all'elemento di `matrice` i cui indici sono individuati dalla posizione corrente `current` il numero corrispondente a quante mosse sono state effettuate, si sceglie a caso una tra le possibili mosse e la si effettua assegnando a `current` la nuova posizione dopo la mossa.

  Se il ciclo riesce a compiere tutte le 64 iterazioni, segnalare che si è trovata una sequenza di mosse che porta al risultato sperato.

Provare ad eseguire alcune volte il programma così costruito e verificare quante mosse in media riesce a compiere il Cavallo prima di non avere più mosse a sua disposizione.

Si può cercare di migliorare il programma precedente introducendo un'"euristica" nel modo in cui il Cavallo effettua le proprie scelte. Analizzando infatti la scacchiera si nota che vi sono caselle che sono meno facilmente raggiungibili rispetto alle altre. Se si assegna ad ogni casella della scacchiera una cifra (un indice di "appetibilità") che rappresenta il numero di posizioni dalle quali è possibile raggiungere la casella fissata, allora per una scacchiera completamente vuota si ottiene la matrice seguente:

$$\begin{pmatrix} 2 & 3 & 4 & 4 & 4 & 4 & 3 & 2 \\ 3 & 4 & 6 & 6 & 6 & 6 & 4 & 3 \\ 4 & 6 & 8 & 8 & 8 & 8 & 6 & 4 \\ 4 & 6 & 8 & 8 & 8 & 8 & 6 & 4 \\ 4 & 6 & 8 & 8 & 8 & 8 & 6 & 4 \\ 4 & 6 & 8 & 8 & 8 & 8 & 6 & 4 \\ 3 & 4 & 6 & 6 & 6 & 6 & 4 & 3 \\ 2 & 3 & 4 & 4 & 4 & 4 & 3 & 2 \end{pmatrix}$$

Mano a mano che il Cavallo esegue delle mosse la matrice precedente deve essere modificata, perché le caselle che sono state toccate devono essere escluse dai conteggi. Utilizzando la matrice precedente si può quindi migliorare la scelta delle mosse: di volta in volta il Cavallo dovrà "preferire" le caselle più difficili da raggiungere, ovvero la scelta non sarà più completamente casuale, ma limitata a quelle caselle che presenteranno indice di appetibilità minimo tra tutte le scelte possibili. Per implementare questo tipo di euristica nel programma precedente si può agire in questo modo:

- si definisce una funzione `aggiorna_accesso` che riceve in input la matrice di configurazione `matrice` e richiama la funzione `mosse_possibili` per costruire la matrice con gli indici di appetibilità;
- all'interno della funzione `gioco`, ad ogni iterazione viene richiamata `aggiorna_accesso` e la scelta sulla mossa da eseguire viene effettuata a caso tra quelle mosse che portano a caselle con indice di appetibilità minimo.

Provare ad eseguire questa variante del programma e verificare se si notano miglioramenti sensibili rispetto alla prima implementazione.