Laboratorio 11 – Esercizi

Elementi di Informatica e Programmazione

Lab 11 – Es 1

Riprendere in esame il Gioco di Nim visto nei precedenti esercizi nim.py e nim2.py e scrivere il programma nim3.py con cui un giocatore umano possa giocare a Nim contro il computer, effettuando una progettazione orientata agli oggetti. Il comportamento del programma deve essere identico al comportamento di nim2.py.

Progettare le **classi** seguenti:

- **NimPile** che rappresenta il mucchio di biglie usato durante una partita. Deve avere:
  - il costruttore che riceve il numero di biglie da inserire inizialmente nel mucchio
  - il metodo **getTotal** che restituisce il numero di biglie presenti nel mucchio
  - il metodo **take** che riceve il numero di biglie da eliminare dal mucchio in seguito a una mossa di uno dei giocatori
- **NimHumanPlayer** che rappresenta il giocatore umano. Deve avere:
  - il metodo **move** che riceve il mucchio di biglie (cioè un oggetto di tipo **NimPile**) e chiede all'utente di fare una mossa (ripetendo la richiesta finché non viene indicata una mossa valida), dopodiché mette in atto la mossa (invocando il metodo **take** del mucchio)
  - dato che gli esemplari di tale classe non hanno bisogno di variabili di esemplare, non hanno bisogno di un costruttore
- **NimComputerPlayer** che rappresenta il computer che gioca. Deve avere:
  - il costruttore che riceve un valore booleano, **True** se e solo se il computer deve giocare in modo *intelligente*
  - il metodo **isExpert** che restituisce **True** se e solo se il computer sta giocando in modo *intelligente*
  - il metodo **move** che riceve il mucchio di biglie (cioè un oggetto di tipo **NimPile**) e calcola la mossa del computer sulla base della modalità in cui sta operando, dopodiché mette in atto la mossa (invocando il metodo take del mucchio)
- **NimGame** che rappresenta un'intera partita. Deve avere:
  - il costruttore che non riceve parametri
  - il metodo **play** che gioca un'intera partita e visualizza il vincitore; se il vincitore è il computer, visualizza anche la modalità di gioco (*intelligente* o *stupido*)

Il programma principale è semplicemente costituito da un ciclo "infinito" che, ad ogni iterazione, crea un esemplare di NimGame, ne invoca il metodo play e, poi, chiede all'utente se vuole fare un'altra partita oppure no.

Lab 11 – Es 2 (1/4)

Scrivere il programma **maze.py** che svolga il ruolo di aiutante nell'uscita da un labirinto.

La classe **Maze** (da progettare) deve rappresentare il labirinto, che è una matrice rettangolare di posizioni, ciascuna delle quali può essere un *corridoio* (di transito) oppure un *muro* invalicabile. Nel labirinto ci si può spostare soltanto da un corridoio a un altro e gli spostamenti possono avvenire soltanto in una delle quattro direzioni cardinali (N=nord, E=est, S=sud, W=ovest), cioè lungo una riga o una colonna della matrice e non in diagonale. Dato un punto di partenza all'interno del labirinto, l'obiettivo è ovviamente quello di arrivare in un corridoio che si trovi sul margine esterno del labirinto. Ciascuna posizione ha 4 posizioni adiacenti (nelle 4 direzioni cardinali), con l'eccezione delle posizioni che si trovano sui margini esterni del labirinto.

Esempio di visualizzazione di un labirinto (usiamo uno spazio per un corridoio e un asterisco per un muro):

```
* *******
*   * * *
* ***** *
* * *   *
* * *** *
*   *   *
***** * *
*   * * *
******* *
```

Lab 11 – Es 2 (2/4)

Ciascuna posizione è individuata da una coppia di indici, di riga e di colonna, numerati a partire da 0 nell'angolo in alto a sinistra e non negativi.

Le informazioni di stato di un oggetto di tipo **Maze** sono memorizzate in un dizionario in cui le chiavi sono tuple che contengono una coppia di indici che individua la posizione di un corridoio (le posizioni che contengono un muro non vengono memorizzate esplicitamente) e i cui valori sono liste di tuple definite allo stesso modo e rappresentanti i corridoi adiacenti (quindi la dimensione di una di queste liste può variare tra 0 e 4).

La classe deve avere i metodi seguenti:

- il costruttore **__init__** deve ricevere il nome di un file che contiene la descrizione di un labirinto nel formato qui visualizzato (asterischi e spazi), per poi leggerlo e creare le informazioni di stato descritte
- il metodo **__repr__** deve restituire una stringa contenente la visualizzazione del labirinto così come illustrato (senza visualizzarla)
- il metodo **help**, quando invocato, deve visualizzare il labirinto con l'aggiunta di informazioni sufficienti per uscire dal labirinto a partire da un corridoio qualsiasi, se da quel punto è possibile farlo, implementando l'algoritmo descritto nel seguito

Il programma deve svolgere alcuni semplici collaudi della classe **Maze**.

Lab 11 – Es 2 (3/4)

Una possibile rappresentazione delle informazioni sufficienti per uscire dal labirinto, trovandosi in un qualsiasi corridoio, è che nel corridoio stesso ci sia scritto quale direzione prendere, cioè verso quale altro corridoio spostarsi. Questa informazione può semplicemente essere una delle 4 lettere che indicano uno dei punti cardinali (oppure un punto interrogativo se da quel punto non è possibile uscire). Ad esempio, il labirinto precedente diventa:

```
*N*******
*NWW*?*S*
*N*****S*
*N*S*EES*
*N*S***S*
*NWW*EES*
*****N*S*
*???*N*S*
*******S*
```

Come possiamo generare queste informazioni all'interno del metodo **help**? Oltre al dizionario che costituisce le informazioni di stato del labirinto, usiamo un altro dizionario che abbia le stesse chiavi (cioè tuple che rappresentano i corridoi) ma, come valori, una delle lettere che possono caratterizzare un corridoio (cioè ?, N, E, S, W). All'inizio, in tutti i corridoi viene scritto ?, dopodiché nei corridoi che si trovano su un margine esterno del labirinto viene scritta la lettera che indica la direzione da seguire per uscire dal labirinto.

Lab 11 – Es 2 (4/4)

Quindi, si inizia un ciclo:

```
done = False
while not done :
  done = True
  per ogni corridoio c nel dizionario delle direzioni
    se c contiene ?
      cerco c nel dizionario delle adiacenze
        ottenendo la lista delle posizioni adiacenti a c
      per ogni posizione p adiacente a c
        cerco p nel dizionario delle direzioni
        se p ha una direzione d diversa da ?
          nel dizionario delle direzioni, il valore associato a c diventa d
          done = False
          break
```

Il metodo help deve visualizzare il labirinto dopo ogni iterazione del ciclo.

Lab 11 – Es 3

Progettare il modulo **triangle.py** che contenga la definizione della classe **Triangle**, i cui esemplari rappresentino triangoli geometrici nel piano cartesiano, oltre a codice di collaudo (ad esecuzione condizionata, come al solito).

Per prima cosa definire la classe **Point** che descriva un punto nel piano cartesiano: il costruttore riceve le coordinate **x** e **y** del punto; il metodo di esemplare **x()** restituisce la coordinata **x** e il metodo di esemplare **y()** restituisce la coordinata **y**, mentre il metodo di esemplare **xy** restituisce una tupla contenente, nell'ordine, la coordinata **x** e la coordinata **y**; il metodo di classe **delta(p1, p2)** calcola e restituisce la distanza tra il punto **p1** e il punto **p2**.

La classe **Triangle** ha un costruttore che richiede tre punti (cioè tre esemplari di **Point**) e solleva l'eccezione **ValueError** se tali punti non costituiscono un triangolo (ricordando che tre punti definiscono un triangolo se e solo se ciascuno dei tre segmenti da essi definiti è minore della somma degli altri due). Per decidere quali siano le migliori informazioni di stato per un tale oggetto, è bene analizzare la sua interfaccia pubblica, così definita:

- il metodo **area()** restituisce l'area del triangolo (può essere utile ricordare la formula di Erone)
- il metodo **height()** restituisce l'altezza relativa al lato maggiore
- il metodo **isScalene()** restituisce **True** se e solo se il triangolo è scaleno
- il metodo **isIsosceles()** restituisce **True** se e solo se il triangolo è isoscele (ovviamente un triangolo equilatero è anche isoscele)
- il metodo **isEquilateral()** restituisce **True** se e solo se il triangolo è equilatero (ovviamente un triangolo equilatero è anche isoscele)
- il metodo **isRight()** restituisce **True** se e solo se il triangolo è rettangolo (ovviamente un triangolo rettangolo può anche essere isoscele o scaleno)

Lab 11 – Es 4 (1/2)

Scrivere il programma **caesarCrypt.py** che cifri un testo usando la cifratura di Cesare: il testo da cifrare è contenuto in un file (sul disco locale o in Internet) il cui nome viene fornito come primo parametro sulla riga dei comandi; il testo cifrato viene visualizzato sul flusso di output standard (e solitamente il programma verrà eseguito utilizzando la redirezione di output); il parametro PARAM che governa la cifratura (che, come vedremo, è un numero intero positivo) può essere fornito come secondo parametro sulla riga dei comandi (e, in sua assenza, viene assunto uguale a 3, così come se tale secondo parametro è errato). Il programma non interagisce con il flusso di input standard.

La cifratura di Cesare opera sostituendo ciascuna lettera del testo (i caratteri che non sono lettere rimangono identici nel testo cifrato) con un'altra lettera determinata procedendo "in avanti" nell'alfabeto di un numero di posti uguale a PARAM, ripartendo dall'inizio se si arriva all'ultima lettera. Ad esempio, con PARAM=3, la lettera A viene sostituita dalla lettera D, B dalla E, e così via, fino a W sostituita da Z, X sostituita da A, Y sostituita da B e Z sostituita da C. Le lettere maiuscole rimangono maiuscole e le lettere minuscole rimangono minuscole. Ovviamente questo algoritmo di cifratura è facilmente decifrabile...

Nota per il collaudo... se PARAM=26 o multiplo, il testo cifrato deve coincidere con il testo da cifrare

Lab 11 – Es 4 (2/2)

Ora, scrivere il programma **caesarDecrypt.py** che decifri un testo che sia stato cifrato usando la cifratura di Cesare: il testo da decifrare è contenuto in un file (sul disco locale o in Internet) il cui nome viene fornito come primo parametro sulla riga dei comandi; il testo decifrato viene visualizzato sul flusso di output standard (e solitamente il programma verrà eseguito utilizzando la redirezione di output); il parametro PARAM che governa la decifrazione può essere fornito come secondo parametro sulla riga dei comandi (e, in sua assenza, viene assunto uguale a 3, così come se tale secondo parametro è errato) e deve (ovviamente...) essere identico a quello utilizzato per la cifratura. Il programma non interagisce con il flusso di input standard.

Per la soluzione, partire dal programma **caesarCrypt.py** e fare il minor numero di modifiche possibili.

Per agevolare il collaudo dei due programmi precedenti, scrivere il programma **fileCompare.py** che confronti il contenuto di due file di testo, i cui nomi vengono forniti come argomenti sulla riga dei comandi. Se i due file sono identici il programma non fornisce alcuna segnalazione, altrimenti scrive "I file sono diversi".

Infine, scrivere il programma **caesarCryptDecrypt.py** che svolga entrambe le funzioni, di cifratura e decifrazione, sulla base del primo parametro fornito sulla riga dei comandi (prima del nome del file): se è **c**, il programma esegue la cifratura; se è **d** il programma esegue la decifrazione; se è una lettera diversa, il programma non fa niente.

Lab 11 – Es 5

Scrivere il programma **vigenere.py** che, usando la cifratura di Vigenère, cifri o decifri un testo sulla base del primo parametro dato sulla riga dei comandi: se è **c**, il programma esegue la cifratura; se è **d** il programma esegue la decifrazione; se è una lettera diversa, il programma non fa niente e termina l'esecuzione. Il testo da cifrare/decifrare è contenuto in un file (sul disco locale o in Internet) il cui nome viene fornito come secondo parametro sulla riga dei comandi; il testo cifrato viene visualizzato sul flusso di output standard (e di solito il programma verrà eseguito usando la redirezione di output); la parola segreta (PASSWORD) che governa la cifratura deve essere fornita come terzo parametro sulla riga dei comandi, altrimenti il programma termina con un messaggio d'errore: la PASSWORD deve contenere solo lettere maiuscole. Il programma non interagisce con il flusso di input standard.

La cifratura di Vigenère cifra ciascuna lettera del testo originario usando un'opportuna cifratura di Cesare, diversa ogni volta. I caratteri che non sono lettere rimangono identici nel testo cifrato. Le lettere maiuscole diventano lettere maiuscole diverse e le lettere minuscole diventano lettere minuscole diverse, con la stessa regola usata per le maiuscole. La specifica cifratura di Cesare da utilizzare per ciascuna lettera dipende dalla PASSWORD. Per codificare la prima lettera del testo si sostituisce la lettera A con la prima lettera della PASSWORD, la lettera B con la lettera dell'alfabeto successiva alla prima lettera della PASSWORD (che, in generale, non sarà la seconda lettera della PASSWORD), e così via. In pratica, è una cifratura di Cesare che usa come PARAM la differenza di posizione, nell'alfabeto, tra la prima lettera della PASSWORD e la lettera A. Per cifrare la seconda lettera del testo si usa lo stesso schema, riferendolo però alla seconda lettera della PASSWORD, e così via. Quando tutte le lettere della PASSWORD sono state utilizzate, si riprende dalla prima lettera della PASSWORD. Quando la lettera della PASSWORD che si sta utilizzando è A, ovviamente la lettera cifrata è identica alla lettera da cifrare.

Nella soluzione, definire una funzione che effettui la codifica/decodifica di un singolo carattere usando la cifratura di Cesare (ricevendo come parametro la "distanza" tra le lettere iniziali dei due alfabeti, il valore PARAM).La decifrazione svolge, ovviamente, la funzione inversa, usando la stessa PASSWORD usata nella cifratura.

Lab 11 – Es 6

Scrivere il programma (non grafico) **tictactoe2.py** che funzioni esattamente come il programma **tictactoe.py** progettato in precedenza (Lab 6 – Es 5), ma usi una classe **TicTacToeBoard** per rappresentare una situazione sulla scacchiera, con le seguenti specifiche:

- il costruttore **__init__(freeChar)** crea una scacchiera vuota, cioè con il carattere **freeChar** in tutte le 9 posizioni; si osservi che nessun metodo della classe deve fare ipotesi sui caratteri utilizzati per rappresentare uno dei due giocatori o una posizione libera
- il metodo **__repr__** restituisce una stringa da usare per visualizzare la scacchiera (deve contenere anche i caratteri per andare a capo al termine di ogni riga)
- il metodo **setCharInPosition(row, column, c)** inserisce il simbolo **c** (che può essere **X** oppure **O**) nella posizione indicata dalle coordinate **row** e **column**
- il metodo **isFree(row, column)** restituisce **True** se e solo se la posizione indicata dalle coordinate **row** e **column** è libera
- il metodo **isFull** restituisce **True** se e solo se la scacchiera è piena, cioè non contiene alcuna posizione libera
- il metodo **isWinner(c)** restituisce **True** se e solo se la scacchiera contiene una configurazione che rende vincitore il giocatore che sta usando il simbolo **c**

Lab 11 – Es 7

Scrivere il programma **maze2.py** che svolga il ruolo di aiutante nell'uscita da un labirinto e costituisca un'evoluzione del programma **maze.py** (proposto in uno dei precedenti laboratori). Il nuovo programma deve generare un labirinto casuale, oltre a poterlo acquisire da un file. La generazione avviene nel costruttore della classe **Maze** che va modificato in modo che possa ricevere il nome di un file o una tupla con i parametri per generare il labirinto, descritti nel seguito.

Rispetto alla versione precedente, al termine del collaudo, nel metodo **help** eliminare la visualizzazione del labirinto ad ogni passo, visualizzandolo soltanto al termine.

Descrizione dell'algoritmo di generazione del labirinto e dei parametri che la regolano:

- i primi due parametri sono la larghezza (**width**) e l'altezza (**height**)
- per prima cosa l'algoritmo costruisce i muri perimetrali del labirinto, che saranno inizialmente completi, senza varchi
- poi, si identificano i varchi da aprire nei muri perimetrali (escludendo i 4 angoli): ogni posizione contenente un muro perimetrale diventa un varco con probabilità espressa dal parametro **holesPercent**, che è un numero nell'intervallo [0, 1] (cioè diventa un varco se e solo se un numero casuale generato con **random()** è minore di **holesPercent**)
- durante tale generazione di varchi, si controlli che ne venga aperto almeno uno: altrimenti, se ne apre uno a scelta
- infine, per ogni posizione interna al rettangolo che delimita il labirinto, si decide se vada costruito un muro che la contiene, cosa che avviene con probabilità espressa dal parametro **wallPercent** (cioè si inizia la costruzione di un muro se e solo se un numero casuale generato con **random()** è minore di **wallPercent**); in caso affermativo, bisogna:
  - decidere casualmente se il muro sarà verticale o orizzontale (probabilità uniforme)
  - decidere casualmente se il muro sarà verso l'alto o verso il basso (nel caso di muro verticale) oppure verso sinistra o verso destra (in caso di muro orizzontale), a partire dalla posizione in esame
  - decidere la lunghezza del muro, scegliendo casualmente (con probabilità uniforme) un numero intero compreso nell'intervallo (1, **wallMaxLength**), dove **wallMaxLength** è un altro parametro di costruzione
  - posizionare il muro, facendo attenzione che non esca dai limiti del rettangolo (nel qual caso va fatto "scivolare" verso la direzione opposta, senza accorciarlo)

Condurre esperimenti per identificare coppie ragionevoli di parametri **wallPercent** e **wallMaxLength**, che sono evidentemente correlati.

Lab 11 – Es 8 (1/2)

Scrivere il modulo **factorGenerator.py** contenente la definizione della classe **FactorGenerator** che effettui la scomposizione di un numero intero positivo nei suoi fattori primi.

La classe deve avere:

- un costruttore che riceve come unico parametro un numero intero e verifica che sia positivo e maggiore di uno (sollevando **ValueError** in caso contrario): sarà il numero da scomporre in fattori primi;
- un metodo di esemplare, **nextFactor**, che non riceve parametri espliciti e che, a ogni sua successiva invocazione, restituisce uno dei fattori primi in cui viene scomposto il numero originariamente fornito nel costruttore; invocando un numero sufficiente di volte tale metodo, si ottengono tutti e soli i fattori primi del numero (eventualmente ripetuti), in modo che moltiplicandoli tra loro si ottenga il numero originario (ad esempio, il numero 150 viene scomposto nei suoi fattori primi 2, 3, 5, 5); se il metodo viene invocato dopo che ha già fornito tutti i fattori primi, esso lancia **StopIteration** (attenzione, non **ValueError**, che non avrebbe senso, dal momento che il metodo non ha parametri espliciti...);
- un metodo di esemplare, **hasMoreFactors**, che non riceve parametri espliciti e che restituisce il valore booleano **True** se e solo se esistono fattori primi non ancora restituiti da **nextFactor** (cioè se **nextFactor** è ancora invocabile).

Porre particolare attenzione all'individuazione delle informazioni di stato necessarie al funzionamento della classe, tentando di minimizzarne la dimensione (*suggerimento*: è sufficiente un'unica variabile di tipo **int**).

Lab 11 – Es 8 (2/2)

Un'alternativa molto più semplice prevede che il costruttore generi una lista contenente tutti i fattori che verranno poi restituiti da **nextFactor**, memorizzando tale lista come variabile di esemplare, insieme a un indice che punta al prossimo elemento della lista che verrà restituito (inizialmente zero). Tale soluzione è vietata, la classe deve avere come unica variabile di esemplare un numero intero.

Dotare il modulo di un programma di collaudo che:

- legge dallo standard input un numero intero maggiore di uno;
- crea un esemplare di **FactorGenerator** fornendo come parametro il numero ricevuto;
- visualizza i fattori primi del numero invocando ripetutamente **nextFactor**.

Infine, usando la classe **FactorGenerator** del modulo **factorGenerator**, scrivere il programma **isPrime.py** che:

- legge dallo standard input un numero intero positivo;
- visualizza sull'uscita standard un messaggio che specifica se il numero introdotto è un numero primo oppure no (ricordare che un numero è primo se i suoi unici fattori primi sono 1 e il numero stesso; ricordare anche che il numero 1 non è primo).

Dopo la fase di lettura del dato in ingresso, il programma non deve avere cicli. La parte difficile dell'esercizio non è scrivere il codice ma capire come usare un oggetto di tipo FactorGenerator per decidere se un numero è primo oppure no, senza eseguire cicli nel programma principale.

Lab 11 – Es 9

Scrivere il programma **maze3.py** che costituisca un'evoluzione del programma **maze2.py** progettato in precedenza. Oltre ad avere tutte le funzionalità di **maze2.py**, il nuovo programma deve poter individuare ed evidenziare il percorso di uscita dal labirinto a partire da una posizione (di corridoio) scelta a caso.

La visualizzazione finale dovrà presentare:

- muri e corridoi come già detto
- punto di partenza (scelto a caso) evidenziato con il carattere **#**
- percorso da seguire evidenziato con i caratteri **+**
- le indicazioni **NESW**, generate nella fase intermedia dal metodo help, non devono essere visibili nella visualizzazione finale
- dopo l'esecuzione di **help**, l'individuazione del percorso avviene nel nuovo metodo **findRoute**, mentre la visualizzazione finale è comunque effettuata da **__repr__**.