Strutture dati e algoritmi e complessità

Esercitazione del 18 Marzo 2025 - Canale A-L

In questa esercitazione si ripropone l'obiettivo dello scorso anno sulla ricorsione, ma si introduce il gioco Salta Casella, semplice esercizio ricorsivo dello stesso livello del Task 3. Chi si sente a suo agio con la ricorsione può saltare il Task 1 (semplice) ed affrontare subito una ricorsione non lineare. I Task 1, 2 e 3 sono presentati usando le stesse parole dell'anno scorso. Il Task 4 aggiunge il Salta Casella.

Scopo di questa esercitazione è risolvere problemi facendo uso di diversi tipi di ricorsione. Si ha ricorsione lineare quando vi è una sola chiamata ricorsiva all'interno della funzione, e non lineare nel caso in cui le chiamate ricorsive siano più di una. Un caso particolare della ricorsione non lineare, o multipla, è quello in cui le chiamate ricorsive sono due (ricorsione binaria).

Attenzione: L'esercitazione può essere svolta sia in linguaggio C che in Java. Entrambe le cartelle sono fornite dello scheletro delle classi da completare e di un programma di prova (`driver.c` e `Driver.java`). Una volta compilato, il programma di prova va invocato con uno dei seguenti argomenti: `inverti` , `maxGap` o `labirinto` , a seconda dell'esercizio che si vuole provare. Nel terzo caso, il programma di prova richiederà di fornire la dimensione del labirinto e poi di specificarlo come una sequenza di stringhe (si veda la descrizione del punto 3)

Non occorre creare nuovi file, ma soltanto completare le classi Java/moduli C forniti. Si consiglia agli studenti di implementare le funzioni C/metodi Java nell'ordine in cui sono proposti.

Task 1. Ricorsione lineare: inverti lista

Si vuole creare un metodo per invertire gli elementi di una lista di interi. Ad esempio, data la lista $L = [10, 20, 5, 6, 8]$ in ingresso, si vuole modificare la lista come segue: $L = [8, 6, 5, 20, 10]$. Inoltre, si vuole che il metodo richieda abbia complessità computazionale $O(n)$ (dove n è il numero di elementi della lista). Il metodo *deve modificare la propria lista di input L*.

Specifiche. Completare il modulo C con intestazione `linked _list.c` (rispettivamente la classe Java `InvertiLista.java`) contenente la funzione (metodo statico se Java):

```
void inverti_lista(linked_list * list)

public static void invertiLista(LinkedList<Integer> list)
```

Task 2. Ricorsione binaria: massimo gap

Si vuole creare un metodo per calcolare il massimo gap presente in un array ordinato di interi. Il gap è definito come la differenza fra due numeri adiacenti nell'array: ad esempio il gap fra 7 e 10, è 3. E' richiesto lo sviluppo di un algoritmo con ricorsione binaria.

Specifiche. Completare il modulo C con intestazione `max_gap.c` (rispettivamente, la classe `maxGap.java`), scrivendo la funzione (metodo statico):

```
int maxGap(int * array, int start, int end)

public static int maxGap(int[] array, int start, int end)
```

Dove `array` è un array di interi con indici $\in [start, end – 1]$. Si proceda a testare `maxGap` utilizzando `driver.c` (rispettivamente `Driver.java`) passando come argomento `maxGap` fornito nella cartella dell'esercitazione.

Suggerimento: per risolvere il problema utilizzando la ricorsione binaria, si consiglia di impostare la soluzione esplorando ricorsivamente la metà destra e sinistra dell'array e poi confrontando i maxGap trovati nei due casi.

Task 3. Ricorsione multipla: risolvi labirinto

Dato un labirinto rappresentato da una matrice quadrata, si vuole trovare un cammino dalla cella di entrata alla cella di uscita. Nel labirinto ci sono celle vuote che possono essere percorse, e celle piene che non possono essere percorse. Il movimento può avvenire in quattro direzioni (*sinistra, destra, alto, basso*). Gli spostamenti in diagonale non sono ammessi. Si rappresenti il labirinto con una matrice `m[][]` di dimensione $n × n$. L'entrata corrisponde alla posizione $(0, 0)$ (angolo in alto a sinistra), e l'uscita alla posizione $(n – 1, n – 1)$ (angolo in basso a destra). Ciascun elemento `m[i][j]` rappresenta una cella vuota o piena (si introducano delle costanti simboliche VUOTA e PIENA, rispettivamente).

Si intende scrivere una classe che determini se esiste un percorso dall'entrata all'uscita del labirinto e, nel caso che esista, lo visualizzi. La lunghezza del cammino trovato non è

rilevante.

Figura 1: Due labirinti 5×5. Il primo ammette soluzione, il secondo no.

La Figura 1 illustra i concetti esposti sopra e le convenzioni usate per rappresentare un labirinto in input al programma.

Suggerimento. Si utilizzi un algoritmo ricorsivo che via via marca – in una matrice ausiliaria `marcata[][]` di $n \times n$ `boolean` – le celle esplorate nel percorso parziale costruito. Si utilizzi il fatto che l'uscita è raggiungibile da una certa cella vuota (i, j) se e solo se essa è raggiungibile da qualche cella vuota adiacente ad (i, j).

Specifiche. Scrivere un modulo C con intestazione `labirinto.c` (rispettivamente una classe Java `Labirinto.java`) con la seguente interfaccia. Si consiglia di implementare i metodi nell'ordine indicato. Si consiglia inoltre di utilizzare una struttura simile alla seguente per rappresentare il labirinto. In C:

```
typedef struct {
    int n;
    int **matrix;
    int **marcata;
} labirinto_struct ;

labirinto * labirinto_new(int n) {
    // Crea un labirinto contenente n x n celle vuote.
    return NULL;
}

void labirinto_delete(labirinto * lab) {
    // TODO: Da completare
}

void labirinto_setpiena(labirinto * lab, int r, int c) {
    // Setta a PIENA la cella m[r][c]
}

int labirinto_uscita(labirinto * lab, int r, int c) {
    // Restituisce true se la cella (r, c) corrisponde all'uscita del la
    return 0;
}

int labirinto_percorribile(labirinto * lab, int r, int c) {
    // Restituisce true se la cella (r, c) è percorribile, dove "percorr
    return 0;
}

int labirinto_uscitaraggiungibileda(labirinto * lab, int r, int c) {
    // Restituisce true se l'uscita del labirinto è raggiungibile dalla
    return 0;
}

int labirinto_risolvibile(labirinto * lab) {
    // Restituisce true se e solo se il labirinto ammette soluzione. In
    return 0;
}

void labirinto_tostring(labirinto * lab, char * buffer, int buffer_size)
    ...
}
```

Si noti che la funzione `labirinto_uscitaraggiungibileda` deve essere ricorsiva.

La funzione `toString` restituisce una stringa (contenuta nel buffer se in C) rappresentante il labirinto, incluso l'eventuale percorso dall'entrata all'uscita (si assuma che il

metodo `risolvibile()` sia già stato invocato prima dell'invocazione di `toString()`). Si rappresenta una cella vuota (non marcata) col carattere '.', una piena col carattere '#', e una cella marcata col carattere '+'. In C, occorre verificare che la dimensione del buffer ricevuto in input sia sufficiente per ospitare la stringa rappresentante il labirinto. L'interfaccia in Java è la seguente, dove le specifiche per i metodi sono le stesse di quelle delle corrispondenti funzioni in C:

```
public class Labirinto {

    private static enum Cella { VUOTA, PIENA };

    private int n;
    private Cella m[][];
    private boolean marcata[][];

  public Labirinto(int n) {
        //TODO: Da Completare
  }

    public void setPiena(int r, int c){
        //TODO: Da Completare
  }

    private boolean uscita(int r, int c){
        //TODO: Da Completare
        return false;
  }

    public boolean percorribile(int r, int c){
        //TODO: Da Completare
        return false;
  }

    private boolean uscitaRaggiungibileDa(int r, int c){
        //TODO: Da Completare
        return false;
    }

    public boolean risolvibile(){
        //TODO: Da Completare
        return false;
    }

    public String toString() {
        //TODO: Da Completare
        return null;
    }
}
```

Si proceda a testare labirinto attraverso il driver (passando come argomento labirinto) fornito nella cartella dell'esercitazione e i labirinti di esempio contenuti nei file di testo `lab1.in` e `lab2.in`. La riga di comando su sistema UNIX:

```
./ driver labirinto < lab1.in
java Driver labirinto < lab1.in
```

invoca il risolutore sul primo esempio.

Task 4. Ricorsione multipla: saltacasella

L'esercizio è descritto nel file `saltacasella.pdf`, presente nella cartella. Il codice di supporto è solo in C ma è pienamente imperativo e può essere immediatamente traslitterato in Java. Non sono necessari tipi specifici, tanto che lo scheletro del codice, inclusivo di `main`, è in un solo file `saltacasella.c`. Si raccomanda il solito comando di compilazione `gcc -g -Wall saltacasella.c -o saltacasella`, che prevede il run eseguendo `./saltacasella <N>`, avendo indicato con `<N>` la dimensione della tabella.