FI2 - Compito di esame A

Fabrizio d'Amore

7 febbraio 2011

Sommario

In questo documento, costruito incrementalmente, verranno presentate delle possibili soluzioni, proposte dal docente, a quesiti d'esame. È fondamentale osservare che si tratta di possibili soluzioni che vengono qui presentate solo per dare supporto agli studenti che hanno difficoltà nella risoluzione autonoma dei quesiti. In particolare, le soluzioni presentate sono specifiche per i quesiti posti e non possono essere automaticamente adottate per rispondere ad altri quesiti, anche se apparentemente simili.

Problema 1

Si consideri il metodo `incrementa(ListIterator<Integer> cifre)` che, dato l'iteratore di una lista che contiene le cifre di generico numero intero (da quella meno significativa a quella più significativa), incrementa di una unità il valore (facendo side effect sulla relativa lista):

```
public static void incrementa(ListIterator<Integer> cifre) {
        if (!cifre.hasNext()) {
            cifre.add(1);
        } else {
            Integer val = cifre.next();//valore corrente
            if (val < 9) {
                cifre.set(val + 1);//assegna un valore all'elemento corrente (costo costante)
            } else {
                cifre.set(0);//assegna il valore 0 all'elemento corrente (costo costante)
                incrementa(cifre);
            }
        }
    }
```

Si richiede quanto segue.

(a) Determinare, giustificandolo opportunamente, il costo computazionale del metodo `incrementa(ListIterator<Integer> cifre)` in funzione della dimensione dell'input.

(b) Scrivere il metodo statico Java `decrementa(ListIterator<Integer> cifre)` che decrementa di una unità il valore rappresentato dall'iteratore (secondo le medesime convenzioni del metodo `incrementa(...)`). Assumere che il valore passato come parametro sia strettamente positivo. Determinare, giustificandolo opportunamente, il costo del metodo.

(c) Scrivere il metodo statico Java

`ListIterator<Integer> differenza(ListIterator<Integer> a, ListIterator<Integer> b)`

che, utilizzando i metodi `incrementa(...)` e/o `decrementa(...)` definiti in precedenza, calcoli e restituisca il valore `a-b` (assumendo `a` $\geq$ `b`). Determinare, giustificandolo opportunamente, il costo del metodo.

Possibile soluzione al Problema 1

(a) Indichiamo con $C(n)$ il costo computazionale dell'algoritmo descritto dal metodo `incrementa`, ove n è il numero di elementi presenti nella lista e, dunque, la dimensione dell'input. Si noti innanzi tutto che tutte le istruzioni del metodo sono caratterizzate dall'avere costo computazionale costante e l'unica fonte di costo non costante deriva dalla chiamata ricorsiva (schema di ricorsione: lineare, con ricorsione di coda). Si tratta dunque di riuscire a calcolare il numero di chiamate ricorsive nel caso peggiore, in funzione di

n. A tal fine, basta notare che a ogni chiamata ricorsiva il metodo lavora su un preciso elemento della lista, ottenuto attraverso `cifre.next()`. Se tale elemento contiene una cifra pari a `9` allora il metodo eseguirà successivamente un'altra chiamata ricorsiva. Il caso peggiore corrisponde alla sequenza di cifre `9999....9999` (n cifre pari a `9`).

Per il costo, che già si intuisce essere pari $\Theta(n)$, possiamo scrivere la seguente (dis)equazione di ricorrenza:

$$C(n) \leq \begin{cases} c' & \text{per } n = 0 \\ C(n-1) + c'' & \text{per } n > 0 \end{cases}$$

ove c' e c'' sono due opportune costanti positive. La ricorrenza si risolve banalmente per "srotolamento" (termine coniato a lezione).

$$C(n) \leq C(n-1) + c'' \leq C(n-2) + 2c'' \leq \cdots \leq C(1) + (n-1)c'' \leq C(0) + nc'' \leq c' + nc''$$

Poiché nel caso peggiore risulta effettivamente $C(n) = c' + nc''$, ne segue che $C(n) \in \Theta(n)$.

(b) Il metodo richiesto è completamente simmetrico a `incrementa` ed è il seguente.

```
public static void decrementa(ListIterator<Integer> cifre) {
  if (!cifre.hasNext())
    return; // qui non dovrebbe mai arrivare
  boolean first; // è il primo elemento della lista?
  if (cifre.hasPrevious()) first = false;
  else first = true;
  Integer val = cifre.next();// valore corrente
  if (val > 0)
    if ((val == 1) && !cifre.hasNext() && !first)
      cifre.remove(); // non vogliamo zeri nelle cifre più
                      // significative, a meno di non aver raggiunto
                      // zero
    else
      cifre.set(val - 1);
  else {
    cifre.set(9);
    decrementa(cifre);
  }
}
```

Il test `!cifre.hasNext()` non dovrebbe mai essere soddisfatto in quanto il testo del problema stabilisce esplicitamente che il numero rappresentato dalla lista fornita in input al metodo `decrementa` è da considerarsi positivo. Il ruolo della variabile `first` è ricordare se il metodo è stato chiamato sul primo elemento della lista: questo è rilevante quando ci si trova a decrementare il numero `1`, (unico) caso in cui il risultato deve mantenere `0` come cifra più significativa.

Il costo computazionale del metodo `decrementa` è pari a $\Theta(n)$ e può essere determinato nella stessa maniera usata per `incrementa`.

(c) Innanzi tutto è bene notare una imprecisione nel testo, che dice di assumere `a` $\geq$ `b`, ma dimentica di specificare che `b` $\geq$ `0`; né potrebbe essere diversamente, poiché non è specificato come la lista di cifre dovrebbe rappresentare un numero negativo.

Il calcolo di `a` $-$ `b` può essere facilmente svolto attraverso `b` decrementi di `a`. Ciò può essere fatto attraverso il semplice ciclo

```
while(b > 0)
  decrementa a
  decrementa b
```

al termine del quale in `a` sarà contenuto il risultato dell'operazione.

Nell'implementare l'algoritmo in Java è necessario tener presente che i due operandi sono specificati attraverso iteratori, per cui occorre prevedere operazioni di "rewind" degli stessi (metodo `restart`).

```
public static ListIterator<Integer> differenza(ListIterator<Integer> a, ListIterator<Integer> b) {
  restart(b);
  while (!isZero(b)) {
    restart(a);
    restart(b);
    decrementa(a);
    decrementa(b);
    restart(b);
  }
  restart(a);
  restart(b);
  return a;
}

private static boolean isZero(ListIterator<Integer> a) {
  while (a.hasNext())
    if (a.next() != 0) return false;
  return true;
}

//reset position to start
private static void restart(ListIterator<Integer> a) {
  while (a.hasPrevious())
    a.previous();
}
```

Gli algoritmi descritti dai metodi `restart` e `isZero` hanno costo $\Theta(n)$ in quanto entrambi prevedono, nel caso peggiore, la scansione dell'intera lista in input.[1] Per quanto riguarda il metodo `differenza`, notiamo che il suo costo è dato dal costo del `while` più i costi dei vari `restart` esterni al ciclo. Questi ultimi sono pari a $\Theta(n)$, essendo n la dimensione dell'input, asintoticamente determinato dal numero di cifre di `a` ($\texttt{a} \geq \texttt{b}$). Per quanto riguarda il ciclo, questo viene eseguito `b` volte ed il costo massimo per ciascuna iterazione è pari a $\Theta(n)$. Poiché il caso $\texttt{a} = \texttt{b}$ massimizza il numero di iterazioni, ne segue che il costo di `differenza` è stimabile come $O(n \cdot \texttt{b}) = O(n \cdot 2^n)$.

Si noti, per questo ultimo costo, l'uso del simbolo 'O' (e non 'Θ'), in quanto l'analisi svolta non tiene conto del fatto che non è possibile che nelle `b` iterazioni si possa sistematicamente verificare il caso peggiore dei metodi `decrementa` e `restart`. L'analisi svolta non è dunque *tight*, ovvero non aderisce al meglio alla realtà computazionale ed è per questo che non può essere usato il simbolo Θ. Nell'ambito del corso di *Fondamenti di informatica II*, in casi come questo non è prevista un'analisi più raffinata.

Problema 2

Con riferimento al tipo astratto Coda di Priorità si richiede quanto segue:

(a) Definire il tipo astratto Coda di Priorità realizzando una possibile interfaccia Java `PQ` che lo descrive.

(b) Realizzare la classe astratta Java `PQListaOrdinata` che implementa l'interfaccia `PQ` e che rappresenta una coda di priorità realizzata mediante lista ordinata contenente chiavi di tipo intero. La classe deve contenere tutte le variabili di istanza e le firme dei metodi fondamentali oltre alla implementazione del metodo di inserimento di un nuovo elemento. Di ogni metodo deve infine essere indicato (e motivato) il costo computazionale.

(c) Definire il metodo `static void PQsort(int[] array)` che ordina il vettore `array` mediante una coda di priorità `PQListaOrdinata`. Indicarne poi, motivandolo adeguatamente, il costo computazionale.

Possibile soluzione al Problema 2

(a) Una coda di priorità è una collezione di elementi, detti valori, a ciascuno dei quali è associata una chiave (o priorità). La coppia chiave-valore viene chiamata entry. Il nome "coda di priorità" deriva dal fatto che le chiavi determinano la priorità usata nel rimuovere le entry.

[1] Per quanto riguarda `isZero` è doveroso precisare che se la lista ha raggiunto il valore 0 attraverso l'uso del metodo `decrementa` proposto al punto precedente, allora il suo costo sarà $O(1)$. Ciò è tuttavia asintoticamente irrilevante nell'analisi in corso.

Le operazioni fondamentali su una coda di priorità sono

- `insert(k, x)`: inserisce il valore `x` con chiave `k` nella coda;
- `removeMin()`: restituisce e rimuove dalla coda una entry avente chiave minima. Nel caso di più entry con la stessa chiave ne viene scelta una qualunque (quale, in pratica, dipenderà dall'implementazione e dall'ordine di inserimento delle entry).

A queste due operazioni fondamentali si aggiunge tipicamente l'operazione `min` che restituisce una entry con chiave minima (la stessa che verrebbe rimossa da `removeMin`), ma senza rimuoverla. Altre operazioni utili sono `size`, che restituisce il numero di entry in coda, e `isEmpty` che restituisce `true` se e solo se la coda è vuota.

```
interface PQ<K,V> {
  public int size(); // restituisce numero di entry nella coda
  public boolean isEmpty(); // restituisce true se e solo se la coda è vuota
  public Entry<K,V> min();  // restituisce una entry con chiave min
  public Entry<K,V> removeMin(); // restituisce e rimuove una entry con chiave min
  public Entry<K,V> insert(K key, V value); // crea la entry (key,value) la inserisce
                                            // in coda e la restituisce
}

interface Entry<K,V> {
  public K getKey();
  public V getValue();
}
```

(b) Si riporta il codice Java richiesto, ove, per ciascun metodo, viene anche indicato il costo asintotico.

```
abstract class PQListaOrdinata<V> implements PQ<Integer, V> {
  private LinkedList<Entry<Integer,V>> listaBase;

  public abstract int size(); // Theta(1) grazie ad analogo metodo su LinkedList

  public abstract boolean isEmpty(); // Theta(1) grazie ad analogo metodo su LinkedList

  public abstract Entry<Integer,V> min();  // Theta(1) perché restituisce testa di LinkedList

  public abstract Entry<Integer,V> removeMin(); // Theta(1) perché rimuove testa di LinkedList

  // Theta(n) perché nel caso peggiore deve scandire tutta la LinkedList
  public Entry<Integer,V> insert(Integer key, V value) {
    Entry<Integer,V> e = new PQEntry<V>(key, value); // Theta(1)
    if(listaBase.isEmpty()) listaBase.add(e); // Theta(1)
    else {
      ListIterator<Entry<Integer,V>> liter = listaBase.listIterator(); // Theta(1)
      boolean done = false; // Theta(1)
      while(liter.hasNext() && !done) // Theta(n) perché può eseguire n iterazioni
        if(liter.next().getKey().compareTo(key) >= 0) { // Theta(1)
          liter.previous();                             // Theta(1)
          liter.add(e);                                 // Theta(1)
          done = true;                                  // Theta(1)
        }
      if(!done) liter.add(e); // inserisce in coda (Theta(1))
    }
    return e; // Theta(1)
  }
}

class PQEntry<V> implements Entry<Integer, V> {
  private Integer key;
  private V value;
```

```
  public PQEntry(Integer key, V value) { // Theta(1) (banale)
    this.key = key;
    this.value = value;
  }

  public Integer getKey() { // Theta(1) (banale)
    return this.key;
  }

  public V getValue() { // Theta(1) (banale)
    return this.value;
  }
}
```

(c) L'algoritmo richiesto è un InsertionSort che, nella forma in cui deve essere specificato, non opera *in place*.

La sua implementazione è immediata e prevede due fasi: nella prima la sequenza da ordinare viene inserita in una coda di priorità; nella seconda, la coda viene svuotata e le chiavi così ottenute costituiscono la sequenza ordinata.

Passando a Java, è necessario osservare che poiché la classe `PQListaOrdinata` è astratta (punto (b)), essa non può essere istanziata e dunque sarà necessario fare riferimento a una classe non astratta `PQListaOrdinataConcreta` che estende `PQListaOrdinata`. Di tale classe, ai fini del quesito d'esame, non è necessario scrivere l'implementazione.

Ai fini dell'ordinamento, interessa inserire nella coda di priorità solo i valori da ordinare (chiavi); ad essi non è necessario associare altre informazioni (`value = null`). Per tale ragione, è possibile costruire una coda di `Entry<Integer, Object>` usando ripetutamente `insert(array[i], null)`.

```
static void PQsort(int[] array) {
  PQListaOrdinata<Object> pq = new PQListaOrdinataConcreta<Object>(); // O(1)
  for(int i = 0; i < array.length; i++) pq.insert(array[i], null); // O(n^2)
  for(int i = 0; i < array.length; i++) array[i] = pq.removeMin().getKey(); // O(n)
}
```

L'algoritmo ha come componente dominante il primo ciclo `for`, che prevede $n =$ `array.length` inserimenti nella coda di priorità `pq`, ciascuno dei quali ha un costo lineare. Poiché all'i-esimo inserimento il costo di `insert` è $\Theta(i)$, il costo globale del primo ciclo `for`, con un lieve abuso di notazione, sarà pari a

$$\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$$

Ragionando in maniera simile, è immediato verificare che il costo del secondo ciclo `for` è pari a

$$\sum_{i=1}^{n} \Theta(1) = \Theta(n)$$

il che conferma che la componente dominante è costituita dal primo ciclo `for`. Il costo dell'algoritmo è pertanto $\Theta(n^2)$.

Problema 3

Con riferimento ai grafi orientati si richiede quanto segue:

(a) Definire il concetto di chiusura transitiva di un grafo.

(b) Definire una possibile rappresentazione per un grafo basata su liste di adiacenza realizzando una classe astratta Java `Graph` contenente tutte le variabili di istanza e le firme di tutti i metodi pubblici che si ritiene fondamentali per risolvere il problema successivo.

(c) Realizzare un metodo di istanza della classe `Graph` la cui firma e' `Graph chiusuraTransitiva()` e che calcola la chiusura transitiva del grafo `this`. Descrivere poi il costo computazionale dell'algoritmo, motivandolo opportunamente.