Esercizi di Algoritmi e Strutture Dati

Moreno Marzolla
marzolla@cs.unibo.it

Ultimo aggiornamento: 10 novembre 2010

1 La "bandiera nazionale"

(problema 4.7 del libro di testo). Il problema della bandiera nazionale italiana è cosí definito. Sia $A[1..n]$ un array i cui elementi possono assumere solo uno di tre possibili valori: *bianco*, *verde* e *rosso*. Vogliamo ordinare l'array in modo che tutti gli elementi verdi precedano quelli bianchi, e gli elementi bianchi precedano quelli rossi. L'algoritmo deve richiedere tempo $O(n)$ nel caso peggiore, può solo scambiare elementi e non deve usare altri array temporanei di appoggio, quindi al più deve richiedere memoria aggiuntiva $O(1)$. Non può nemmeno utilizzare contatori per tenere traccia del numero di elementi di un certo colore presenti (quindi non si può usare una variante dell'algoritmo *counting sort*, che vedremo a lezione). L'algoritmo, infine, deve ordinare gli elementi facendo una singola scansione dell'array. Assicurarsi che l'algoritmo funzioni anche se mancano elementi di qualcuno dei colori.
Suggerimento: ripensare a come funziona la procedura *partition* dell'algoritmo Quick Sort, e provare ad adattarla per risolvere questo problema.

```
/*
 * Bandiera.java - questo programma risolve il problema 4.7 p. 116 di
 * Demetrescu, Finocchi, Italiano, "Algoritmi e strutture dati"
 * (seconda edizione), McGraw-Hill, 2008.
 *
 * Il problema della bandiera nazionale e' definito nel modo seguente:
 * Sia A un array di dimensione n, i cui elementi possono assumere
 * solo uno di tre possibili colori: bianco, verde e rosso. Vogliamo
 * ordinare l'array in modo che tutti gli elementi verdi precedano
 * quelli bianchi, e gli elementi bianchi precedano quelli
 * rossi. L'algoritmo deve richiedere tempo O(n) nel caso peggiore,
 * puo' solo scambiare elementi e non deve usare altri array
 * temporanei di appoggio. Non puo' nemmeno utilizzare contatori per
 * tenere traccia del numero di elementi di un certo colore presenti
 * (quindi non si puo' usare una variante dell'algoritmo counting
 * sort, che vedremo a lezione). L'algoritmo, infine, deve ordinare
 * gli elementi facendo una singola scansione dell'array. Assicurarsi
 * che l'algoritmo funzioni anche se mancano elementi di qualcuno dei
 * colori.
 *
```

```
 * Version 0.3 del 2009/11/25
 * Autore: Moreno Marzolla (marzolla (at) cs.unibo.it)
 *
 * This file has been released by the author in the Public Domain
 */
public class Bandiera {

    public static final int VERDE = 1;
    public static final int BIANCO = 2;
    public static final int ROSSO = 3;

    /**
     * Scambia tra di loro gli elementi A[i] e A[j]. Attenzione:
     * nessun controllo viene effettuato circa la validita' degli
     * indici i e j.
     */
    protected static void swap( int[] A, int i, int j )
    {
        System.out.println("Scambia␣"+i+"␣con␣"+j);
        int tmp = A[i];
        A[i] = A[j];
        A[j] = tmp;
    }

    /**
     * Stampa a video il contenuto dell'array A
     */
    public static void stampa( int[] A )
    {
        for( int i=0; i<A.length; ++i ) {
            System.out.print(A[i]);
            System.out.print("␣");
        }
        System.out.println();
    }

    /**
     * Risolve il problema della bandiera nazionale: ordina l'array
     * A[] (che puo' contenere solo i valori 1=VERDE, 2=BIANCO,
     * 3=ROSSO) in una unica passata.
     *
     * Questo metodo ha complessita' Theta(n).
     */
    public static void ordina( int[] A )
    {
        int n = A.length; // numero elementi de
        int v = 0; // Prima posizione in cui si puo' inserire un elemento verde
        int i = 0; // Prima posizione in cui si puo' inserire un elemento bianco
        int r = A.length-1; // Prima posizione in cui si puo' inserire un elemento rosso

        /**
         * L'algoritmo mantiene le seguenti invarianti:
         *
         * 1. Gli elementi A[0]...A[v-1] sono verdi (se v=0, non ci
         * sono elementi verdi);
         *
         * 2. Gli elementi A[v]..A[i-1] sono bianchi (se i==v, non ci
```

```
         * sono elementi bianchi);
         *
         * 3. Gli elementi A[r+1]..A[n-1] sono rossi (se r==n-1, non
         * ci sono elementi rossi);
         *
         * 4. Gli elementi A[i]...A[r] possono essere di qualsiasi colore
         *
         * Ad ogni iterazione, l'algoritmo esamina A[i], effettuando
         * opportuni scambi per spostare l'elemento nella posizione
         * corretta. Ad ogni iterazione, i viene incrementato OPPURE r
         * viene decrementato. Cio' assicura che sia garantita la
         * terminazione dell'algoritmo.
         */
        while( i<= r ) {
            if ( BIANCO == A[i] ) {
                i++;
            } else {
                if ( VERDE == A[i] ) {
                    swap( A, v, i );
                    v++;
                    i++;
                } else { // A[i] e' rosso
                    swap( A, i, r );
                    r--;
                }
            }
        }
    }

    public static void main( String[] args )
    {
        int p1[] = {1, 2, 3, 1, 3, 1, 2, 3, 3, 1, 1, 2, 3, 1, 1 };
        Bandiera.ordina(p1);
        Bandiera.stampa(p1);
        int p2[] = {1, 1, 1, 1};
        Bandiera.ordina(p2);
        Bandiera.stampa(p2);
        int p3[] = {2, 2, 2, 2};
        Bandiera.ordina(p3);
        Bandiera.stampa(p3);
        int p4[] = {3, 3, 3, 3};
        Bandiera.ordina(p4);
        Bandiera.stampa(p4);
        int p5[] = {3, 1, 1, 3, 1, 3};
        Bandiera.ordina(p5);
        Bandiera.stampa(p5);
        int p6[] = {3, 3, 3, 2};
        Bandiera.ordina(p6);
        Bandiera.stampa(p6);
    }
}
```

2 Un problema apparentemente complicato

(problema 4.8 del libro di testo). Descrivere un algoritmo che dato un array $A[1..n]$ di interi appartenenti all'insieme $\{1, 2, \ldots k\}$, preprocessa l'array in tempo $O(n + k)$ in modo da generare una opportuna struttura dati che consenta di rispondere in tempo $O(1)$ a query del tipo: "quanti elementi di A sono compresi nell'intervallo $[a, b]$?" (per qualsiasi $1 \leq a \leq b \leq k$)

```
/*
 * Intervalli.java - questo programma risolve il problema 4.9 p. 116
 * di Demetrescu, Finocchi, Italiano, "Algoritmi e strutture dati"
 * (seconda edizione), McGraw-Hill, 2008.
 *
 * Descrivere un algoritmo che dato un array A di n numeri interi,
 * tutti compresi nell'intervallo [1,k], preprocessa l'array in tempo
 * O(n+k) in modo da generare una opportuna struttura dati che
 * consenta di rispondere in tempo O(1) a query del tipo: ``quanti
 * elementi di $A$ sono compresi nell'intervallo [a,b]?'' (per
 * qualsiasi $1 \leq a \leq b \leq k$)
 *
 * Version 0.1 del 2009/11/09
 * Autore: Moreno Marzolla (marzolla (at) cs.unibo.it)
 *
 * This file has been released by the author in the Public Domain
 */
public class Intervalli {

    int sum[]; // sum[i] e' il numero di elementi di A che sono minori
               // o uguali di i, i=1,2,...k. sum[0] vale zero
    int k;

    /**
     * Preprocessa l'array A[] (che puo' contenere esclusivamente
     * valori interi compresi nell'intervallo [1,...k]) in modo da
     * poter poi rispondere in tempo O(1).
     *
     * Questo metodo ha complessit' Theta(n+k), essendo n il numero di
     * elementi in A[].
     */
    public Intervalli( int A[], int k )
    {
        this.k = k;
        int i;
        sum = new int[k+1];
        // Initializza sum[] a zero
        for ( i=0; i<k+1; ++i )
            sum[i] = 0;
        // Calcola i valori si sum[] iterando una sola volta sull'array A[]
        for ( i=0; i<A.length; ++i ) {
            sum[A[i]]++;
        }
        // Calcola le somme
        for ( i=1; i<k+1; ++i ) {
            sum[i] = sum[i-1]+sum[i];
        }
```

```
    }

    /**
     * Restituisce il numero di elementi di A[] compresi tra a e b
     * (estremi inclusi). Questo metodo ha complessit' O(1).
     */
    public int conta( int a, int b )
    {
        // si assume che 1 <= a <= b <= k
        return (sum[b] - sum[a-1]);
    }

    public static void main( String args[] )
    {
        int A[] = {1, 2, 3, 1, 3, 2, 2, 4, 2, 3, 1, 3, 2, 2, 4, 1, 1, 1, 2};
        Intervalli interv = new Intervalli(A,4);
        System.out.println("Numero elementi in [1,2] = " + interv.conta(1,2));
        System.out.println("Numero elementi in [2,3] = " + interv.conta(2,3));
        System.out.println("Numero elementi in [3,4] = " + interv.conta(3,4));
    }
}
```

3 L'attitudine al problema vs l'attitudine alla soluzione

(problema 4.11 del libro di testo). Progettare un algoritmo che, dato in input un array $A[1..n]$ contenente n valori reali, restituisca in tempo $O(n)$ un *qualsiasi numero* che **non** appartenga ad A. Dimostrare che $\Omega(n)$ è un limite inferiore al costo computazionale di questo problema.

Soluzione Visto che qui si richiede di individuare un numero *qualsiasi* che non appartiene all'insieme dato, si può procedere nel modo seguente:

- Si calcola il minimo x dell'array (questa operazione ha costo $\Theta(n)$);
- Si restituisce come risultato il valore $x-1$, che sicuramente non appartiene ad A.

È immediato che l'algoritmo così fatto ha complessità $\Theta(n)$. Per quanto riguarda il limite inferiore, è sufficiente considerare che per individuare un numero che non appartenga al vettore dato, è *sempre* necessario esaminare ciascun elemento del vettore almeno una volta. Se i valori non venissero tutti presi in considerazione, ci sarebbe sempre la possibilità che il valore restituito possa essere presente tra quelli non esaminati. Da ciò si conclude the $\Omega(n)$ è un limite inferiore alla complessità di questo problema.

4 Verifica Min-Heap

Quesito presente nel primo parziale dell'Anno Accademico 2009/2010 Scrivere un algoritmo efficiente per risolvere il seguente problema: dato un array $A[1..n]$ di $n > 0$ valori reali, restituire *true* se l'array A rappresenta un min-heap binario, *false* altrimenti. Calcolare il costo computazionale nel caso pessimo e nel caso ottimo dell'algoritmo proposto, motivando le risposte.

Soluzione Una possibile soluzione è la seguente:

```
Algoritmo isMinHeap( Array A[1..n] )
  for i:=1 to n do
    if ( 2*i <= n && A[i] > A[2*i] ) then
      return false;
    endif
    if ( 2*i+1 <= n && A[i] > A[2*i+1] ) then
      return false;
    endif
  endfor
  return true;
```

Si notino i controlli `(2*i<=n)` e `(2*i+1<=n)`: tali controlli sono **indispensabili** per essere sicuri che i figli dell'elemento $A[i]$ (rispettivamente $A[2 * i]$ e $A[2 * i + 1]$) siano effettivamente presenti nell'array. Se tali controlli non sono effettuati, l'algoritmo è da considerarsi **sbagliato** in quanto può causare l'indicizzazione di un array al di fuori dei limiti.

Il caso pessimo si ha quando il ciclo "for" viene eseguito interamente, ossia quando l'array $A[1..n]$ effettivamente rappresenta un min-heap. In questo caso il costo è $\Theta(n)$. Il caso ottimo si verifica quando la radice $A[1]$ risulta maggiore di uno dei due figli ($A[2]$ o $A[3]$, se esistono). In questo caso il ciclo "for" esce alla prima iterazione restituendo false, e il costo risulta $O(1)$. In definitiva, si può dire che il costo di questo algoritmo è $O(n)$.

5 Modifica di Selection Sort

Quesito presente nel primo parziale dell'Anno Accademico 2009/2010 Considerare la seguente variante dell'algoritmo `selectionSort()`:

```
Algoritmo mioSelectionSort(array A[1..n])
  for k:=0 to n-2 do
    m:=k+1;
    for j:=k+2 to n do
      if (A[j] <= A[m]) then
        m := j;
      endif
    endfor
```

```
    scambia A[k+1] con A[m];
  endfor
```

Ricordiamo che un algoritmo di ordinamento è *stabile* se preserva l'ordinamento relativo degli elementi uguali presenti nell'array da ordinare. L'algoritmo `mioSelectionSort()` è un algoritmo di ordinamento stabile? In caso negativo, mostrare come modificarlo per farlo diventare stabile. Motivare le risposte.

Soluzione L'algoritmo `mioSelectionSort()` è *quasi* identico all'algoritmo `SelectionSort()` descritto nel libro di testo, con la differenza che la condizione stata scritta come `if ( A[j] <= A[m] )` (usando l'operatore $\leq$ anziché $=$). Questo fa sì che `mioSelectionSort` **non** sia un algoritmo di ordinamento stabile, perché in caso di elementi uguali mette in prima posizione l'ultimo elemento trovato. Per renderlo stabile sufficiente riscrivere l'if come `if ( A[j] < A[m] )`