

# Soluzioni di Algoritmi e Strutture Dati

Ultimo aggiornamento: 25 novembre 2009

Moreno Marzolla  
marzolla@cs.unibo.it

6 Novembre 2009

## 1 Esercizi sulla complessità asintotica

### 1.1 Vero o falso?

Per ciascuna delle seguenti affermazioni, dire se è vera o falsa, fornendo una dimostrazione:

1.  $3^n = O(2^n)$
2.  $2^{2n} = O(2^n)$
3.  $2^{n+1} = O(2^n)$
4.  $\log_3 n = O(\log_2 n)$
5.  $\ln n = O(n^\alpha)$ , per ogni  $\alpha > 0$  ( $\ln$  è il logaritmo in base  $e$ ).

#### Soluzione

1. Proviamo a verificare se esiste una costante  $c > 0$  tale che  $3^n \leq c2^n$  per  $n \geq n_0$ . Dividendo entrambi i membri per  $2^n$  si richiederebbe che  $(3/2)^n \leq c$  per una qualche costante  $c$ . Poiché  $\lim_{n \rightarrow +\infty} (3/2)^n = +\infty$ , si ha che la relazione di cui sopra non può essere verificata per  $n$  arbitrariamente grande. Concludiamo quindi che l'affermazione è **falsa**.

2. Come sopra, proviamo a verificare se esiste una costante  $c > 0$  tale che, per valori sufficientemente grandi di  $n$ , valga:

$$2^{2n} \leq c2^n$$

Si noti che  $2^{2n} = (2^2)^n = 4^n$ . Seguendo la stessa argomentazione del punto precedente, si richiede che  $4^n \leq c2^n$  per una qualche costante  $c$ . Concludiamo quindi che l'affermazione è **falsa**.

3. Proviamo a verificare se esiste una costante  $c > 0$  tale che, per valori sufficientemente grandi di  $n$ , valga:

$$2^{n+1} \leq c2^n$$

Si noti che  $2^{n+1} = 2 \times 2^n$ , per cui possiamo scrivere

$$2 \times 2^n \leq c2^n$$

che è certamente verificata ponendo ad esempio  $c = 2$ , per ogni  $n \geq 0$ . Quindi l'affermazione è **vera**.

4. Verifichiamo se  $\log_3 n \leq c \log_2 n$ . Sia  $x = \log_3 n$ . Dalla definizione di logaritmo significa che  $3^x = n$ . Prendendo il logaritmo in base 2 di entrambi i membri, otteniamo:

$$\log_2(3^x) = \log_2 n$$

da cui

$$x \log_2 3 = \log_2 n$$

e quindi, ricordando che  $x$  era  $\log_3 n$  si ha:

$$\log_3 n = \frac{\log_2 n}{\log_2 3}$$

In generale, per ogni  $y > 0$ ,  $b, c > 1$  vale la proprietà di cambio di base dei logaritmi (descritta in appendice nel libro di testo) per cui

$$\log_a y = \frac{\log_b y}{\log_b a}$$

Tornando al problema iniziale, si tratta ora di trovare una costante  $c > 0$  tale per cui

$$\frac{\log_2 n}{\log_2 3} \leq c \log_2 n$$

che è verificata se  $c \geq 1/\log_2 3$ . Quindi l'affermazione è **vera**.

5. Verifichiamo se  $\ln n \leq cn^\alpha$ , per una opportuna costante  $c > 0$  e per  $n$  sufficientemente grande. Per fare questo, studiamo il limite seguente:

$$\lim_{n \rightarrow +\infty} \frac{\ln n}{n^\alpha}$$

Derivando numeratore e denominatore (regola di de l'Hôpital) si ottiene

$$\lim_{n \rightarrow +\infty} \frac{\ln n}{n^\alpha} = \lim_{n \rightarrow +\infty} \frac{1/n}{\alpha n^{\alpha-1}} = \lim_{n \rightarrow +\infty} \frac{1}{\alpha n^\alpha} = 0$$

Dalla definizione di limite, possiamo concludere che l'affermazione  $\ln n = O(n^\alpha)$  è **vera**.

## 1.2 Dimostrazioni per induzione

Il seguente esercizio mostra come sia particolarmente importante prestare attenzione a come si fanno le dimostrazioni per induzione. Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} 2T(\lfloor n/2 \rfloor) + 1 & n > 1 \\ 1 & n = 1 \end{cases}$$

È relativamente facile rendersi conto che  $T(n) = O(n)$ . Riuscite a dimostrare per induzione che  $T(n) \leq cn$ , per una opportuna costante  $c > 0$ ? Riuscite a dimostrare che  $T(n) \leq cn - b$ , per opportune costanti  $c > 0$  e  $b$  arbitraria?

**Soluzione** Proviamo a dimostrare che  $T(n) \leq cn$  per una opportuna costante  $c > 0$ . Il caso base  $T(1) = 1 \leq c$  è valido per qualunque  $c \geq 1$ . Vediamo ora il passo induttivo:

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + 1 \\ &\leq 2c(n/2) + 1 \\ &= cn + 1 \end{aligned}$$

A questo punto però **non** possiamo proseguire affermando  $cn + 1 \leq cn$ , perché questo non sarebbe vero. Il problema si risolve usando  $cn - b$  al posto di  $cn$  nella dimostrazione sopra. Infatti, in questo caso il passo induttivo risulta essere

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + 1 \\ &\leq 2(c(n/2) - b) + 1 \\ &= cn - 2b + 1 \\ &\leq cn - b \end{aligned}$$

che risulta essere verificato se  $b \geq 1$ . Il caso base:

$$T(1) = 1 \leq c - b$$

risulta verificato scegliendo  $c \geq b + 1$ , il che completa la dimostrazione.

### 1.3 Problema con il caso base

Il seguente esercizio mostra un esempio in cui si presenta un problema con il caso base, facilmente aggirabile. Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} 2T(\lfloor n/2 \rfloor) + n & n > 1 \\ 1 & n = 1 \end{cases}$$

Dimostrare per induzione che  $T(n) = O(n \log n)$  (ossia, dimostrare che  $T(n) \leq cn \log n$  per una opportuna costante  $c > 0$  e per  $n > n_0$ ).

**Soluzione** Partiamo alla rovescia, dal passo induttivo:

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\leq 2c(n/2) \log(n/2) + n \\ &= cn \log(n/2) + n \\ &= cn(\log n - 1) + n \\ &= cn \log n - cn + n \\ &= cn \log n + n(1 - c) \\ &\leq cn \log n \end{aligned}$$

l'ultima equazione vale se  $c > 1$ .

Vediamo ora il caso base:

$$T(1) = 1 \not\leq c \log 1 = 0$$

Fortunatamente, in questi casi possiamo sfruttare a nostro vantaggio il fatto che la relazione  $T(n) \leq cn \log n$  non deve necessariamente valere a partire da  $n = 1$ , ma solo per  $n \geq n_0$  con  $n_0$  scelto opportunamente. Quindi il caso base della dimostrazione per induzione non necessariamente deve essere 1, ma possiamo sceglierlo pari al primo valore  $n_0$  per cui la relazione vale. Quindi:

$$\begin{aligned} T(2) &= 2T(1) + 2 = 4 \leq c2 \log 2 \\ T(3) &= 3T(1) + 3 = 6 \leq c3 \log 3 \\ T(4) &= 2T(2) + 4 \end{aligned}$$

È quindi possibile individuare un valore di  $c > 1$  che soddisfi le prime due equazioni.

## 2 Definizione e analisi di algoritmi

**Nota:** nel caso sia richiesto di descrivere un algoritmo, è possibile usare pseudocodice, oppure un frammento di codice Java, a scelta. Il codice Java non deve necessariamente compilare, è sufficiente che descriva completamente l'algoritmo richiesto. Come suggerimento personale, consiglio di utilizzare lo pseudocodice in modo da non perdere tempo sui dettagli della sintassi del linguaggio. Nel caso si usi pseudocodice, prestare particolare attenzione a specificare in modo dettagliato *tutti* i passi eseguiti dall'algoritmo. Nel caso di Java si assume sempre che l'indice degli array parte da zero. Nel caso dello pseudocodice si può assumere che parta da uno (ma è necessario specificarlo chiaramente).

### 2.1 Caso pessimo di Quick Sort

Consideriamo l'algoritmo Quick Sort nella versione deterministica vista a lezione, in cui il valore del pivot è sempre scelto come il valore del primo elemento dell'array da partizionare. Quando si verifica il caso pessimo per questa versione dell'algoritmo?

**Soluzione** Quando l'array è già ordinato.

### 2.2 La “bandiera nazionale”

(*problema 4.7 del libro di testo*). Il problema della bandiera nazionale italiana è così definito. 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, può solo scambiare elementi e non deve usare altri array temporanei di appoggio. 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 0.2 del 2009/11/12
 * 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);
}
}

```

### 2.3 Un problema apparentemente complicato

(problema 4.8 del libro di testo). Descrivere un algoritmo che dato un array  $A$  di  $n$  numeri interi, tutti compresi nell'intervallo  $[1, k]$  ( $k > 1$ ), 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 $$$ 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];
    // Inizializza sum[] a zero
    for ( i=0; i<k+1; ++i )
        sum[i] = 0;
    // Calcola i valori di 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 di elementi in [1,2] = " + interv.conta(1,2));
    System.out.println("Numero di elementi in [2,3] = " + interv.conta(2,3));
    System.out.println("Numero di elementi in [3,4] = " + interv.conta(3,4));
}
}

```

## 2.4 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 di  $n$  numeri reali, trovi in tempo lineare un *qualsiasi* numero che **non** appartenga all'array. Dimostrare poi che  $\Omega(n)$  è un limite inferiore al numero di passi necessari per risolvere 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 sta nell'array dato.

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 all'insieme dato, è *sempre* necessario esaminare ciascun numero dell'insieme almeno una volta. Se i valori non venissero tutti presi in considerazione, ci sarebbe sempre la possibilità che il valore restituito potesse essere presente tra quelli non esaminati. Da ciò si conclude che  $\Omega(n)$  è un limite inferiore alla complessità di questo problema.