Appunti di Algoritmi e Strutture Dati

Nicola Canzonieri

Ottobre 2024

Indice

1 Capitolo 1

1.1 Introduzione e Problemi di ordinamento

I problemi di ordinamento consistono nell'ordinare una data sequenza di interi, contenuta in un vettore di interi A di lunghezza n.
Vedremo una serie di algoritmi che si occuperanno di svolgere questo problema e in particolare andremo a studiare la **complessità spaziale, temporale** e la **correttezza** di questi algoritmi.

1.2 Insertion Sort

L'idea alla base dell'algoritmo Insertion Sort è quella che dato un vettore A dove sappiamo che $A[1...i-1]$ è già ordinato, per ordinare tutto $A[1...i]$ ci basterà inserire $A[i]$ nel posto giusto.
Possiamo applicare questa strategia pensando già da subito che il vettore $A[1...1]$ è già ordinato e che quindi ci basterà iniziare a posizionare $A[2]$ e così via.

Questo è lo pseudocodice di Insertion Sort:

```
InsertionSort(A) {
        for (i <- 2 to n) {
                key <- A[i]
                j <- i - 1

                while (key < A[j] and j > 0) {
                        A[j + 1] <- A[j]
                        j <- j - 1
                }

                A[j + 1] <- key
        }
}
```

1.3 Complessità temporale di Insertion Sort

Analizziamo adesso la complessità temporale di Insertion Sort andando a costruire la funzione che ci ritornerà il tempo di esecuzione che l'algoritmo necessita:

`for (i <- 2 to n)` necessita tempo $c_1 + c_2(n-1)$

`key <- A[i]` necessita tempo $c_3(n-1)$

`j <- i - 1` necessita tempo $c_4(n-1)$

`while (key < A[j] and j > 0)` necessita tempo $c_5 \sum_{i=2}^{n} t_i$ dove t_i indica il numero di volte che l'intestazione viene eseguita durante la i-esima iterazione del `for`

`A[j + 1] <- A[j]` necessita tempo $c_6 \sum_{i=2}^{n}(t_i - 1)$

`j <- j - 1` necessita tempo $c_7 \sum_{i=2}^{n}(t_i - 1)$

`A[j + 1]` necessita tempo $c_8(n-1)$

Sommando tutti i termini c_i abbiamo che:

$$T_{Insert}(n) = c_1 + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n}(t_i - 1) + c_7 \sum_{i=2}^{n}(t_i - 1) + c_8(n-1)$$

Che possiamo semplificare in:

$$T_{Insert}(n) = a + b(n-1) + c\sum_{i=2}^{n} t_i$$

Da questa nuova equazione (molto più maneggiabile) possiamo analizzare il tempo necessario per la computazione dell'algoritmo Insertion Sort nel **caso peggiore** e nel caso **caso minore**.

Per quanto riguarda il **caso peggiore** possiamo subito dire che questo caso corrisponde al caso in cui il vettore è ordinato in ordine decrescente e quindi avremo che $t_i = i$ da cui vediamo che:

$$\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$$

e quindi:

$$T_{Insert}(n) = a + b(n-1) + c\frac{n(n+1)}{n} - c = \Theta(n^2)$$

che vuol dire che la complessità temporale nel caso peggiore è **quadratica**.

Nel **caso migliore** il vettore è già ordinato e quindi $t_i = 1$ poiché l'intestazione del while viene eseguita comunque una volta per il confronto. Quindi abbiamo che:

$$T_{Insert} = a + b(n-1) + c(n-1) = \Theta(n)$$

che vuol dire che la complessità temporale nel caso migliore è **lineare**.

1.4 Complessità spaziale e stabilità di Insertion Sort

Lo spazio richiesto dall'algoritmo (cioè il numero di variabili presenti) non dipende dalla grandezza del vettore A (infatti le variabili sono sempre `i`, `j` e `key`). Per questo motivo abbiamo che:

$$S_{Insert}(n) = \Theta(1)$$

che vuol dire che la complessità spaziale è **costante** e quindi si dice anche che Insertion Sort è un algoritmo **in place**.

Un altro concetto molto importante è quello della **stabilità**, infatti un algoritmo di ordinamento che fa si che gli elementi ripetuti restino nello stesso ordine viene detto **algoritmo stabile**.

1.5 Correttezza di Insertion Sort

Dobbiamo dimostrare che $\forall A$, `InsertionSort(A)` termina con A ordinato. Questo corrisponde al nostro **enunciato di correttezza**.

Per procedere abbiamo bisogno di un **invariante** per il ciclo `for` che risponderà alla domanda: "Alla i-esima iterazione del ciclo `for`, cos'è sempre vero?".
La risposta sarà: "All'inizio della i-esima iterazione del `for`, $A[1...i-1]$ è ordinato".
Dimostriamolo per induzione su i:

Caso base: i = 2
$A[1...i-1] = A[1...1]$ è ordinato poiché è composto da un unico elemento

Passo induttivo
Supponiamo che alla i-esima iterazione $A[1...i-1]$ sia ordinato, e dimostriamo che $A[1...i]$ verrà ordinato alla $i+1$-esima iterazione.
Analizzando il corpo del ciclo `for` vediamo che il ciclo `while` sposta a dx gli elementi senza modificare l'ordine. Infatti tutti i valori prima del j-esimo (una volta che il ciclo while si è interrotto) sono ordinati e la stessa cosa vale per quelli dopo.
Per questo motivo il valore che verrà inserito nel j-esimo posto sarà maggiore di tutti quelli prima di lui e minore di tutti quelli dopo di lui.
Inoltre poiché la guardia del ciclo for (vale a dire i) non è stata toccata all'interno del ciclo stesso possiamo essere sicuri che il for terminerà all'inizio della $(n+1)$-esima iterazione per l'invariante avremo che $A[1...(n+1)-1] = A[1...n]$ è ordinato.

2 Esercizio Algoritmo del massimo

Facciamo un analisi uguale a quella vista prima ma con un algoritmo di ricerca del massimo in un vettore.

Vediamo lo pseudocodice dell'algoritmo:

```
Min(A) {
        min <- A[1]

        for (i <- 2 to A.length) {
                if (min > A[i]) {
                        min <- A[i]
                }
        }

}
```

2.1 Complessità

Recuperiamo la funzione che ci ritornerà il tempo usato dall'algoritmo:

`min <- A[1]`
c_1

`for (i <- 2 to A.length)`
$c_2 + c_3(n-1)$

`if (min > A[i])`
$c_4(n-1)$

`min <- A[i]`
$c_5(n-1)$

Quindi abbiamo che:

$$T_{Max}(n) = c_1 + c_2 + c_3(n-1) + c_4(n-1) + c_5(n-1)$$

che possiamo semplificare in:

$$T_{Max}(n) = a + b(n-1) = \Theta(n)$$

In questo caso non abbiamo fatto un'analisi del caso peggiore e del caso migliore poiché la complessità temporale non cambia.

2.2 Correttezza

Per dimostrare la correttezza dobbiamo dimostrare l'invariante del ciclo for che nel nostro caso è quello che all'i-esima iterazione, `max` è uguale al valore massimo del vettore