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 e 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, allora ci basterà inserire $A[i]$ nel posto giusto in modo che il vettore $A[1...i]$ sia così ordinato.
Possiamo applicare questa strategia pensando già da subito che il vettore $A[1...1]$ è già ordinato 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 sintetizzare 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 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**.