Algoritmi di ordinamento

(Si può fare meglio di così?)

Emanuele Nardi

Compilato il 3 febbraio 2019
v1.0.0

```
countingSort(ITEM[] A, int n, int k)
    int i, j, k
    int[ ] B ← new int[1...k]

    // Inizializza i vettori
    per i ← 1 fino a k fai B[i] = 0
    per j ← 1 fino a n fai B[A[j]] = B[A[j]] + 1

    j = 1
    da i = 1 fino a k fai
        finché B[i] > 0 fai
            A[j] ← i
            j++
            B[i] ← B[i] − 1
```

$$\mathcal{O}(n + k)$$

Algoritmo 1.1 – Algoritmo di ordinamento

```
// efficiente per ordinare piccoli insiemi di elementi
insertionSort(ITEM[] A, int n)

    da int i = 2 fino a n fai // il 1° elemento è ordinato
        ITEM temp ← A[i] // elemento da ordinare
        int j ← i

        finché j > i and A[j − 1] fai
            A[j] ← A[j − 1] // copio l'elemento
            j ← j − 1 // mi sposto

        A[j] ← temp

// vettore già ordinato:  Ω(n)
// vettore decrescente:  O(n^2)
// in media O(n^2)
```

Algoritmo 1.1 – Algoritmo di ordinamento D&I

```
mergeSort(Item[] A, int primo, int ultimo)
    se primo < ultimo allora // devono esistere almeno due elementi
        int mezzo ← ⌊(primo+ultimo)/2⌋
        mergeSort(A, primo, mezzo)
        mergeSort(A, mezzo+1, ultimo)
        merge(A, primo, ultimo, mezzo)

merge(Item A, int primo, int ultimo, int mezzo)
    int i, j, k, h
    // Inizializzo i puntatori
    i ← primo j ← mezzo k ← primo
    // k:  indica la prossima posizione di scrittura
    // fintanto che entravi
    finché i ⩽ mezzo and j ⩽ ultimo fai
        se A[i] ⩽ A[j] allora
            // l'elemento è gia ordinato
            B[k] ← A[i]
            i++
        altrimenti
            B[k] ← A[j]
            j++
        // in entrambi i casi ho inserito un valore
        k++
    // se uno dei due vettori finisce ricopio la parte ordinata alla fine del vettore
       d'appoggio
    j ← ultimo
    da h ← mezzo fino a i fai
        A[j] ← A[h]
        j −−
    // ricopio il vettore d'appoggio del vettore originale
    da j ← primo fino a k − 1 fai
        A[j] ← B[j]
```

Equazione di ricorrenza:

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

Analisi per livelli:

$$\mathcal{O}\left(\sum_{i=0}^{k} 2^i \frac{n}{2^i}\right) = \mathcal{O}\left(\sum_{i=0}^{k} n\right) = \mathcal{O}(k \cdot n) = \mathcal{O}(n \log n)$$

Teorema dell'esperto:

$$\alpha = \log_2 2 = 1 \qquad \beta = 1 \qquad \alpha = \beta$$

$$T = \mathcal{O}(n^{\alpha} \log n) = \mathcal{O}(n \log n)$$

Estensione del countingSort che permette di ordinare in tempo lineare coppie (chiave, valore), invece che singoli interi. Le chiavi devono essere comprese fra 1 e k.

```
// ordina un vettore di RECORD in base al campo numerico key associato ad ognuno di essi
pigeonholeSort(RECORD[] A, int n, int min, int max)
    int size ← max − min + 1
    List() L ← new List[0 . . . size − 1]
    da j ← 1 fino a size fai
        L[j] ← List
    // scansione iniziale
    da i ← 1 fino a n fai
        List M ← L[A[i].key − min]
        M.insert(M.tail, A[i])
    i = 1
    // scansione vettore B
    da j ← 0 fino a size − 1 fai
        Pos p ← L[j].head
        finché not L[j].finished(p) fai
            A[i] ← L[j].read(p)
            i++
            p ← L[j].next
```

```
quickSort(Item[] A, int primo, int ultimo)
    se primo < ultimo allora
        int j ← perno(A, primo, ultimo)
        quickSort(A, primo, j − 1)
        quickSort(A, j + 1, ultimo)

int perno(Item[] A, int primo, int ultimo)
    Item x ← A[primo]
    int j ← primo
    da i ← primo fino a ultimo fai
        se A[i] < x allora
            j++
            swap(A[i], A[j])
    A[primo] ← A[j]
    A[j] ← x
    ritorna j
```

Algoritmo 1.1 – Algoritmo di ordinamento

```
selectionSort(Item[ ] A, int n)
    da int i ← 1 fino a n fai
        int j ← min(A, i, n)
        swap(A[i], A[j])

int Item[ ] A
    int min ← k // posizione del minimo parziale
    da int h ← k + 1 fino a n fai
        se A[h] < A[min] allora
            min ← h // nuovo minimo parziale
```

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

```
shellSort(Item[] A, int n)
    int h ← 1
    finché h ⩽ n fai
        int h ← 3 · h + 1
    int h ← ⌊h/3⌋
    finché not h ⩾ 1 fai
        da i ← h + 1 fino a n fai
            Item temp ← A[i]
            int j ← i
            finché j > h and A[j − h] > temp fai
                A[j] ← A[j − h]
                j ← j − h
            A[j] ← temp
        int h ← ⌊h/3⌋
```