Algoritmi di ordinamento lineari

Un ordinamento di chiavi può talvolta essere effettuato senza operare confronti di chiavi. In questi casi non vale la limitazione inferiore di complessità $\Omega(n \log n)$ che si era trovata assumendo confronti di chiavi. Infatti, sotto opportune ipotesi, la complessità può abbassarsi fino a $O(n)$.

1 – Integer sort

Si consideri il caso in cui vi sono n chiavi tutte distinte con valori in $\{1, \ldots, n\}$, oppure associabili in modo biunivoco agli interi $\{1, \ldots, n\}$. Si può obiettare che con tali chiavi non c'è nemmeno bisogno di ordinarle perché è noto a priori il loro ordine. Non sempre però gli oggetti da ordinare sono 'virtuali'. Se si tratta di oggetti reali, il problema può consistere nel disporli secondo l'ordine noto. Ad esempio un mazzo di carte rientra in questo caso, in quanto possiamo associare ad ogni carta un valore fra 1 e 52. Se vi è una corrispondenza biunivoca fra le chiavi ed i numeri da 1 a n, basta predisporre un vettore di dimensione n e poi assegnare ad ogni chiave la sua posizione corrispondente nel vettore. Nell'esempio del mazzo di carte si possono associare i numeri da 1 a 13 per le picche, da 14 a 26 per i cuori, da 27 a 39 per i quadri e da 40 a 52 per i fiori, si predispone un vettore di 52 posizioni e poi ogni carta del mazzo (disordinato) viene a turno sistemata al suo posto.

È chiaro che basta una scansione delle chiavi, per cui la complessità è $O(n)$. Tuttavia l'algoritmo non è in loco in quanto bisogna disporre di una memoria aggiuntiva pari al numero delle chiavi. Si può modificare l'algoritmo rendendolo in loco con una leggera spesa sul tempo, ma rimanendo in una complessità lineare. Si prende la prima chiave, la si sistema al suo posto (nel vettore delle chiavi) dopo aver prelevato la chiave che vi si trovava. Poi si procede ricorsivamente con le varie chiavi, finché si trova un chiave che si posiziona nel posto della prima chiave. Questo corrisponde ad aver trovato uno dei cicli che definisce la permutazione delle chiavi. A questo punto si procede ad esaminare una chiave alla volta finché non si trova una chiave in posizione errata. Quando la si trova si riparte con la procedura di riposizionamento che terminerà necessariamente nella stessa posizione. Nel caso peggiore sono richieste due scansioni delle chiavi, una quando la chiave viene sistemata al posto giusto e la seconda per vedere se si trova al posto giusto.

Si consideri ora una variante un po' più complicata, in cui le n chiavi assumono valori in $\{1, \ldots, m\}$ e non sono necessariamente distinte. In questo caso bisogna preventivamente contare la molteplicità delle chiavi. Questo si effettua tramite un vettore ausiliario Y, con m posizioni, e con una scansione delle chiavi. Al termine della scansione il vettore contiene il numero delle copie di ogni chiave. Con una scansione di Y possiamo produrre le chiavi ordinate.

```
procedure simpleintegersort(K)
    { p := 1;
    while p ≤ |K| do
        { while p ≤ |K| ∧ k[p] = p do p := p + 1;
        q := k[p];
        while q ≠ p do
            { h := k[q];
            k[q] := k[p];
            q := h
            }
        p := p + 1;
        }
    } .

procedure integersort(K)
    {
    for i := 1 to m do Y[i] := 0;
    for i := 1 to |K| do Y[k[i]] := Y[k[i]] + 1;
    p := 1;
    for i := 1 to m do
        for j := 1 to Y[j] do { X[p] := i, p := p + 1 }
    } .
```

2 – Bucketsort

Una variante più complicata (e anche più interessante) si ha quando dati diversi possono avere la stessa chiave. In questo caso contare quante volte è presente la chiave stessa non basta perché la chiave non individua più univocamente il dato. Bisogna necessariamente memorizzare in posizioni adiacenti dati con la stessa chiave. L'algoritmo integersort va quindi modificato definendo $Y[i]$ come una lista di dati, anziché come un intero. Il nuovo algoritmo, che riceve in input la lista X dei dati e le rispettive chiavi K, è

```
procedure bucketsort(X, K)
    { for k := 1 to m do { Y[k] := ∅; p[k] := 1 };
    for i := 1 to |X| do {Y[K[i], p[K[i]]] := X[i]; p[K[i]] := p[K[i]] + 1 };
    p := 1;
    for k := 1 to m do
        for j := 1 to p[k] − 1 do { Z[p] := Y[k, j], p := p + 1 } ;
    return(Z)
    } .
```

Se, ad esempio i dati X sono le date di nascita di un certo numero di persone e le chiavi sono i dodici mesi dell'anno, l'algoritmo bucketsort restituisce i dati in ordine di mese di nascita e, all'interno dello stesso mese di nascita, nello stesso ordine con cui sono inizialmente. Quest'ultima proprietà è importante e merita una definizione:

Definizione: *Un algoritmo di ordinamento si dice* stabile *se preserva l'ordine fra due dati che hanno la stessa chiave.* ∎

La complessità di bucketsort è data dalle tre scansioni, la prima per creare il vettore di liste Y, la seconda per scrivere il vettore Z e la terza per produrre come output il vettore Z (la terza potrebbe anche essere fatta contestualmente alla seconda). La prima scansione ha costo $O(n)$, la seconda ha costo $O(m+n)$ e la terza $O(n)$. Quindi la complessità è $O(n+m)$.

3 – Radix Sort

L'algoritmo bucketsort non può essere usato per valori elevati di m, come si vede dall'espressione della complessità computazionale. Ad esempio se dovessimo ordinare 100 numeri con valori compresi fra 1 e 10.000, dovremmo costruire il vettore di liste Y con ben 10.000 elementi e poi scandirlo tutto (anche se poi risulterà in gran parte vuoto).

Se i dati presentano un ordine lessicografico, ovvero sono stringhe di simboli di un alfabeto piccolo e l'ordine si basa sui simboli e sulla posizione dei simboli nella stringa, si può applicare ripetutamente bucketsort per ogni elemento della stringa prendendo come insieme di chiavi solo l'alfabeto dei simboli.

Se ad esempio i dati sono dei numeri, allora l'alfabeto è dato dalle dieci cifre. Immaginiamo di applicare bucketsort prendendo come chiave di ogni numero la cifra meno significativa, cioè quella delle unità. Siano, ad esempio, da ordinare i numeri

$$\{6234, 4783, 354, 8957, 134, 2674, 1383, 7228, 5504, 4951, 67, 5376\}$$

Si ottiene, applicando bucketsort alle cifre dell'unità

$$\{4951, 4783, 1383, 6234, 354, 134, 2674, 5504, 5376, 8957, 67, 7228\}$$

Poi si applica bucketsort al nuovo insieme prendendo come chiave la cifra delle decine ottenendo

$$\{5504, 7228, 6234, 134, 4951, 354, 8957, 67, 2674, 5376, 4783, 1383\}$$

La nuova lista è ovviamente ordinata rispetto alle decine, ma anche, grazie alla proprietà di stabilità, alle unità. Infatti se si guardano solo le due ultime cifre decimali, i numeri sono ordinati.

Ora è chiaro cosa bisogna fare, applicare bucketsort prendendo come chiavi le cifre delle centinaia, ottenendo

$$\{67, 134, 7228, 6234, 354, 5376, 1383, 5504, 2674, 4783, 4951, 8957\}$$

e alle cifre delle migliaia, ottenendo

$$\{67, 134, 354, 1383, 2674, 4783, 4951, 5376, 5504, 6234, 7228, 8957\}$$

Per valutare la complessità sia m il numero delle chiavi possibili. Quindi nell'esempio $m = 10.000$ e sia k il numero di chiavi su cui opera bucketsort ($k = 10$ nell'esempio). La relazione fra m e k è $m = k^p$. Il costo dell'ordinamento è dato dal costo di ogni bucketsort ($O(n+k)$) volte il numero di esecuzioni di bucketsort ($O(p) = O(\log_k m)$), quindi $O((n+k)\ \log_k m)$. Se manteniamo k costante la complessità è $O(n \log m)$.

Questo risultato non sembra molto soddisfacente, perché, se $m < n$ conviene usare direttamente bucketsort con complessità $O(n + m) = O(n)$ e, se invece $m > n$, $O(n \log m)$ è peggiore degli ordinamenti con confronto di chiavi. Si può tuttavia scegliere k in modo opportuno in modo da ridurre la complessità.

Immaginiamo di ripetere l'ordinamento dei numeri dell'esempio, ma di farlo usando bucketsort direttamente su cento chiavi, anziché dieci. Quindi nella prima passata i dati vengono direttamente ordinati in base alle due cifre meno significative, producendo

$$\{5504, 7228, 6234, 134, 4951, 354, 8957, 67, 2674, 5376, 4783, 1383\}$$

e saltando così una passata rispetto alla esecuzione precedente. Una seconda passata poi fornisce l'ordinamento finale. In questo esempio ogni esecuzione di bucketsort costa 10 volte tanto e metà delle esecuzioni di bucketsort è stata risparmiata. In totale quindi avere scelto $k = 100$ anziché 10 non è vantaggioso.

Possiamo però chiederci quale sia il valore di k più vantaggioso. Calcoliamo allora $\min_k (n + k) \log_k m$. Derivando rispetto a k e ponendo la derivata uguale a zero, si ottiene (ricordando che $\log_k m = \ln m / \ln k$)

$$\frac{\ln m}{\ln k} - (n + k) \frac{\ln m}{k (\ln k)^2} = 0 \quad \Longrightarrow \quad k \ln k = n + k$$

Questa espressione suggerisce di prendere $k = \Theta(n)$, ad esempio $k = \alpha\, n$ con α costante prefissata, per cui $O((n + k) \log_k m)$ diventa

$$O(n\,(1 + \alpha) \log_k m) = O(n\,(1 + \alpha) \frac{\ln m}{\ln(\alpha\, n)}) = O(n \frac{\ln m}{\ln n})$$

Questa formula va infine corretta per tener conto del caso $m < n$ in cui bisogna spendere tempo almeno $O(n)$ per leggere i dati. Quindi radix sort può essere fatto eseguire con complessità

$$O(n + n \frac{\ln m}{\ln n})$$