Azzolini Riccardo 2019-04-02

Metodi di ordinamento digitali

1 Metodi digitali

I metodi digitali non sono basati su confronti e scambi come operazioni fondamentali, ma accedono ai bit dei dati per determinare come riorganizzarli.

Di conseguenza, possono avere complessità nel caso peggiore inferiore al limite $n \log n$, che riguarda solo gli algoritmi della classe confronti e scambi.

2 Distribution counting

Se l'intervallo dei valori contenuti in una sequenza è limitato, è possibile effettuare l'ordinamento in tempo lineare calcolando la frequenza assoluta di ciascun valore e "ricostruendo" la sequenza in base alle frequenze.

```
public class DistributionCounting {
    public static void sort(int[] a) {
        int N = a.length;

        // Calcola il range di valori
        int min = a[0];
        int max = a[0];
        for (int i = 1; i < N; i++) {
            if (a[i] < min) min = a[i];
            else if (a[i] > max) max = a[i];
        }
        int r = max - min + 1;

        int[] count = new int[r];
        for (int i = 0; i < r; i++) count[i] = 0;

        // Calcola le frequenze assolute
        for (int i = 0; i < N; i++) count[a[i] - min]++;
        // Trasforma le frequenze da assolute a cumulate
        for (int i = 1; i < r; i++) count[i] += count[i - 1];
```

```
        // Ricostruisci la sequenza ordinata
        int[] b = new int[N];
        for (int i = 0; i < N; i++) {
            int freq = count[a[i] - min]--;
            b[freq - 1] = a[i];
        }

        for (int i = 0; i < N; i++) a[i] = b[i];
    }
}
```

1. Si trovano il valore minimo e il massimo, e li si usa per calcolare la dimensione `r` del range di valori.
2. Si crea il vettore `count` di `r` contatori per le frequenze, e si inizializzano i suoi elementi a 0.
3. Si calcolano le *frequenze assolute* (cioè il numero di occorrenze di ogni valore): in base al valore di ogni elemento `a[i]`, si incrementa il contatore corrispondente, `a[i] - min`.
4. Si trasformano le frequenze assolute in *frequenze cumulate*, cioè, per ogni valore, il numero di elementi minori o uguali a esso: a ogni contatore di `count` viene aggiunta la somma di quelli precedenti. Dopo questo passaggio, `count[a[i] - min]` $= f$ significa che `a` contiene f valori $\leq$ `a[i]`.
5. Si copia ogni valore di `a` in un nuovo vettore `b` (della stessa lunghezza), determinando la posizione in cui inserirlo in base alla frequenza cumulata. Ad esempio, se `a[i]` $= 24$ e ci sono 120 valori ≤ 24, si inserisce 24 in posizione $120 - 1 = 119$ (perché gli indici partono da 0) e si decrementa la frequenza cumulata (nel vettore `count`) da 120 a 119.
6. Si ricopiano i dati da `b` ad `a`.

2.1 Complessità

- Nell'algoritmo sono presenti vari cicli, ciascuno con un numero fisso di iterazioni a costo costante: alcuni ne eseguono $\Theta(N)$, altri $\Theta(r)$. Perciò, il numero complessivo di operazioni è $\Theta(N + r)$.
- Oltre a quello occupato dalla sequenza, lo spazio necessario è $\Theta(N + r)$: $\Theta(N)$ per il vettore `b` e $\Theta(r)$ per `count`.

Di conseguenza, quest'algoritmo non è adatto all'ordinamento di sequenze anche corte, ma che contengono valori appartenenti a un range molto ampio (ad esempio, 10 interi compresi tra 0 e 2^{64}).

2.2 Esempio

La sequenza `a = [2,2,1,1,3,1,2,3]` contiene valori da `min = 1` a `max = 3`, con frequenze assolute `count = [3,3,2]`, e frequenze cumulate `count = [3,6,8]`.

Viene quindi costruito il vettore `b`:

		count	b
i	a[i]	0 1 2	0 1 2 3 4 5 6 7
		[3,6,8]	[, , , , , , ,]
0	2	[3,5,8]	[, , , , ,2, ,]
1	2	[3,4,8]	[, , , ,2,2, ,]
2	1	[2,4,8]	[, ,1, ,2,2, ,]
3	1	[1,4,8]	[,1,1, ,2,2, ,]
4	3	[1,4,7]	[,1,1, ,2,2, ,3]
5	1	[0,4,7]	[1,1,1, ,2,2, ,3]
6	2	[0,3,7]	[1,1,1,2,2,2, ,3]
7	3	[0,3,6]	[1,1,1,2,2,2,3,3]

Infine, la sequenza ordinata `b = [1,1,1,2,2,2,3,3]` viene ricopiata in `a`.

3 Ordine lessicografico

Siano

- $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ un **alfabeto** (totalmente) ordinato: $\sigma_1 < \sigma_2 < \cdots < \sigma_k$;
- $\Sigma^\star = \bigcup_{i \geq 0} \Sigma^i$ l'insieme delle **parole** (di qualsiasi lunghezza $i \geq 0$) su Σ;
- $x, y \in \Sigma^\star$ due parole su Σ;
- $|x|$ la lunghezza di x.

x è un **prefisso** di y se e solo se $\exists z \in \Sigma^\star$ tale che $y = xz$.

x è minore di y in base all'**ordine lessicografico**, $x \leq_{\text{lex}} y$, se e solo se

- x è un prefisso di y, oppure
- $\exists \alpha, u, v \in \Sigma^\star,\ \sigma, \tau \in \Sigma$ tali che $x = \alpha\sigma u$, $y = \alpha\tau v$ e $\sigma < \tau$, cioè, tolto il prefisso comune α, la lettera iniziale della parte rimanente di x è minore rispetto a quella della parte rimanente di y.

4 Bucket sort

L'algoritmo **bucket sort** ordina lessicograficamente una sequenza di parole della stessa lunghezza mediante confronti tra singoli caratteri. Esso accede quindi a porzioni specifiche dei dati, e per questo si considera un algoritmo digitale.

Il procedimento di base si definisce induttivamente, data una sequenza $(x_1, \dots, x_n)$:

- Se $|x_i| = 1$, cioè se ciascuna parola è formata da un singolo carattere, si costruisce la sequenza ordinata $(x_{j_1}, \dots, x_{j_n})$ in base all'ordine definito su Σ.
- Altrimenti, $|x_i| > 1$ e, di conseguenza, ogni parola x_i può essere scomposta in un carattere iniziale σ_{j_i} seguito da un suffisso y_i non nullo:

  $$x_i = \sigma_{j_i} y_i, \quad |y_i| > 0$$

  Allora, si costruisce la sequenza ordinata dei suffissi, $(y_{j_1}, \dots, y_{j_n})$, poi si aggiungono i caratteri iniziali, ottenendo una nuova sequenza $(\sigma_{j_1} y_{j_1}, \dots, \sigma_{j_n} y_{j_n})$, la quale viene successivamente ordinata rispetto al primo carattere con un metodo stabile, in modo che, quando tale carattere è uguale, venga mantenuto l'ordine dei suffissi.

  La sequenza ottenuta alla fine è $(\sigma_{l_1} y_{l_1}, \dots, \sigma_{l_n} y_{l_n})$, le cui parole sono in ordine lessicografico, dato che $\sigma_{l_i} \le \sigma_{l_{i+1}}$ e $\sigma_{l_i} = \sigma_{l_{i+1}} \implies y_{l_i} \le y_{l_{i+1}}$.

```java
public static void bucketsort(DEQueue<String> l) {
    DEQueue<String>[] s = new DEQueue<String>[NSIMB];
    for (int j = WORD_LENGTH - 1; j >= 0; j--) {
        while (!l.isEmpty()) {
            String w = l.front(); l.delete(1);
            int e = digit(w, j);
            s[e].enqueue(w);
        }
        for (int i = 1; i < NSIMB; i++) {
            s[0].chain(s[i]); s[i] = null;
        }
        l = s[0]; s[0] = null;
    }
}
```

- `DEQueue` è una "double ended queue", cioè una coda che consente accesso in tempo $O(1)$ sia al primo che all'ultimo elemento. Di fatto, è una lista con riferimenti al primo e all'ultimo nodo, che consentono di effettuare in tempo costante operazioni come `enqueue` e `chain` (la concatenazione).
- `NSIMB` è la cardinalità (numero di simboli) dell'alfabeto.

- `s` è un vettore di `DEQueue` (“bucket”), ciascuna associata a un simbolo dell’alfabeto. È omessa l’inizializzazione delle singole `DEQueue`.
- `WORD_LENGTH` è la lunghezza delle parole da ordinare.
- `digit(w, j)` restituisce la posizione (a partire da 0) del carattere `w.charAt(j)` nell’alfabeto Σ.
- Gli assegnamenti `s[i] = null` e `s[0] = null` rappresentano lo svuotamento delle rispettive `DEQueue`.

Per ogni indice `j`, dall’ultimo al primo, delle lettere da confrontare, il ciclo `for` esterno esegue un’iterazione, nella quale:

1. Il ciclo `while` interno distribuisce le parole dalla lista `l` alle liste `s`:
   a) estrae una parola `w` da `l`;
   b) determina l’indice `e` della lista corrispondente alla `j`-esima lettera di `w`;
   c) accoda la parola `w` alla lista `s[e]`.
2. Le liste `s` vengono concatenate in ordine alfabetico. In questo modo, si costruisce un’unica lista ordinata stabilmente rispetto alla `j`-esima lettera. Nel frattempo, le liste vengono svuotate per l’iterazione successiva.
3. La lista ottenuta dalle concatenazioni diventa quella da ordinare all’iterazione successiva, cioè in base al carattere `j` $-$ 1.

4.1 Esempio

Dato l’alfabeto $\Sigma = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$, si vuole ordinare la lista

$$L = (\mathrm{BABA}, \mathrm{DACA}, \mathrm{BACC}, \mathrm{ADDA})$$

Siccome ogni parola è composta da 4 lettere, sono necessarie 4 iterazioni del ciclo `for` esterno:

- $j = 3, \quad L_1 = (\mathrm{BAB\underline{A}}, \mathrm{DAC\underline{A}}, \mathrm{BAC\underline{C}}, \mathrm{ADD\underline{A}})$

$$\begin{aligned} L_A &= (\mathrm{BABA}, \mathrm{DACA}, \mathrm{ADDA}) \\ L_B &= \Lambda \\ L_C &= (\mathrm{BACC}) \\ L_D &= \Lambda \end{aligned}$$

- $j = 2, \quad L_2 = (\mathrm{BA\underline{B}A}, \mathrm{DA\underline{C}A}, \mathrm{AD\underline{D}A}, \mathrm{BA\underline{C}C})$

$$\begin{aligned} L_A &= \Lambda \\ L_B &= (\mathrm{BABA}) \\ L_C &= (\mathrm{DACA}, \mathrm{BACC}) \\ L_D &= (\mathrm{ADDA}) \end{aligned}$$

- $j = 1, \quad L_3 = (\mathrm{B\underline{A}BA}, \mathrm{D\underline{A}CA}, \mathrm{B\underline{A}CC}, \mathrm{A\underline{D}DA})$

$$\begin{aligned} L_A &= (\mathrm{BABA}, \mathrm{DACA}, \mathrm{BACC}) \\ L_B &= \Lambda \\ L_C &= \Lambda \\ L_D &= (\mathrm{ADDA}) \end{aligned}$$

- $j = 0, \quad L_4 = (\mathrm{\underline{B}ABA}, \mathrm{\underline{D}ACA}, \mathrm{\underline{B}ACC}, \mathrm{\underline{A}DDA})$

$$\begin{aligned} L_A &= (\mathrm{ADDA}) \\ L_B &= (\mathrm{BABA}, \mathrm{BACC}) \\ L_C &= \Lambda \\ L_D &= (\mathrm{DACA}) \end{aligned}$$

Si ottiene così la lista

$$(\mathrm{ADDA}, \mathrm{BABA}, \mathrm{BACC}, \mathrm{DACA})$$

che è ordinata lessicograficamente.

4.2 Analisi

Il bucket sort è stabile, ma non adattivo.

Data una sequenza di n parole, ciascuna di lunghezza k, su un alfabeto di cardinalità m:

- la complessità temporale è $\Theta(k(n + m))$, poiché il ciclo `for` esterno esegue k iterazioni, ciascuna costituita da:
  - $n = \Theta(n)$ iterazioni a costo costante del ciclo `while` interno;
  - $m - 1 = \Theta(m)$ iterazioni a costo costante del ciclo `for` interno;
  - altre istruzioni a costo costante;
- lo spazio aggiuntivo necessario è $\Theta(n + m)$:
  - il vettore `s` contiene m riferimenti a `DEQueue`;
  - le `DEQueue` contengono complessivamente tutte le n parole alla fine del ciclo `while` interno.

A differenza del distribution counting, quindi, il bucket sort si può applicare anche quando il range di valori della sequenza è grande, perché la complessità non dipende da quest'ultimo.

La complessità è lineare nel numero di parole, ma in pratica si ottengono spesso prestazioni migliori con gli algoritmi ottimali della classe confronti e scambi, la cui complessità è $O(n \log n)$. Ad esempio, per ordinare una lista di interi a 32 bit

- usando i singoli bit come caratteri, si ha $k = 32$ (e $m = 2$), quindi un algoritmo con complessità $O(n \log n)$ ha prestazioni migliori finché $\log n < 32 \iff n < 2^{32}$ (almeno in teoria);
- interpretando gli interi in base 16, cioè raggruppando i bit a 4, si ha $k = 8$ (con $m = 16$), cioè un algoritmo $O(n \log n)$ è più veloce solo per $\log n < 8 \iff n < 2^8$, ma per parole di lunghezze superiori può non essere comunque sufficiente applicare questo tipo di "trucchi".