Azzolini Riccardo 2019-04-03

Divide et impera e merge sort

1 Merging

Problema: **Merging** (*a due vie*)

- *Input*: due sequenze ordinate $V, W \in U^\star$.
- *Output*: una sequenza ordinata contenente tutti gli elementi di V e W.

In base alla rappresentazione delle sequenze, si ha il merging di vettori o di liste.

In generale, il merging di due sequenze di lunghezza n_1 e n_2 richiede tempo $O(n_1+n_2)$.

1.1 Esempio

$$
\begin{aligned}
V &= (1, 4, 6, 9) \\
W &= (5, 7, 11, 20, 50)
\end{aligned}
$$

V	W	Z
$(\underset{\uparrow}{1}, 4, 6, 9)$	$(\underset{\uparrow}{5}, 7, 11, 20, 50)$	$()$
$(1, \underset{\uparrow}{4}, 6, 9)$	$(\underset{\uparrow}{5}, 7, 11, 20, 50)$	(1)
$(1, 4, \underset{\uparrow}{6}, 9)$	$(\underset{\uparrow}{5}, 7, 11, 20, 50)$	$(1, 4)$
$(1, 4, \underset{\uparrow}{6}, 9)$	$(5, \underset{\uparrow}{7}, 11, 20, 50)$	$(1, 4, 5)$
$(1, 4, 6, \underset{\uparrow}{9})$	$(5, \underset{\uparrow}{7}, 11, 20, 50)$	$(1, 4, 5, 6)$
$(1, 4, 6, \underset{\uparrow}{9})$	$(5, 7, \underset{\uparrow}{11}, 20, 50)$	$(1, 4, 5, 6, 7)$
$(1, 4, 6, 9)$	$(5, 7, \underset{\uparrow}{11}, 20, 50)$	$(1, 4, 5, 6, 7, 9)$
$(1, 4, 6, 9)$	$(5, 7, 11, 20, 50)$	$(1, 4, 5, 6, 7, 9, 11, 20, 50)$

Osservazioni:

- Le due sequenze possono avere lunghezze diverse.

- Una sequenza si può esaurire prima dell'altra. In tal caso, gli elementi rimasti nell'altra sequenza si trasferiscono senza confronti.

2 Merging di vettori

```
public static void merge(Comparable[] a, int lo, int mid, int hi) {
    Comparable[] aux = new Comparable[a.length];
    for (int k = lo; k <= hi; k++) aux[k] = a[k];

    int i = lo;
    int j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) a[k] = aux[j++];
        else if (j > hi) a[k] = aux[i++];
        else if (less(a[j], a[i])) a[k] = aux[j++];
        else a[k] = aux[i++];
    }
}
```

Quest'implementazione esegue il merge di due sequenze contenute nello stesso vettore `a`: la prima è `a[lo..mid]`, cioè dalla posizione `lo` alla posizione `mid`, mentre la seconda è `a[mid+1..hi]`.

Osservazione: Quando entrambe le sequenze contengono lo stesso valore, cioè `aux[i] == aux[j]`, viene trasferito per primo quello della prima sequenza, `aux[i]`, in modo che l'algoritmo sia stabile (rispetto all'ordine originale degli elementi nel vettore `a`).

2.1 Complessità

Se le due sequenze hanno rispettivamente lunghezza n_1 e n_2:

- lo spazio necessario (oltre alle sequenze) è $\Theta(n_1 + n_2)$, per il vettore `aux`;
- il numero di confronti eseguiti è $\Theta(n_1 + n_2)$ nel caso peggiore, e $\Theta(1)$ nel caso migliore, quando una delle due sequenze contiene solo il valore minimo, quindi si esaurisce immediatamente e l'altra sequenza viene copiata senza confronti;
- quando una delle due sequenze si esaurisce, non sono più necessari confronti, ma bisogna comunque ricopiare tutti gli elementi da `aux` ad `a`, quindi il numero di operazioni eseguite (tempo di calcolo) è $\Theta(n_1 + n_2)$ in ogni caso.

3 Merging di liste

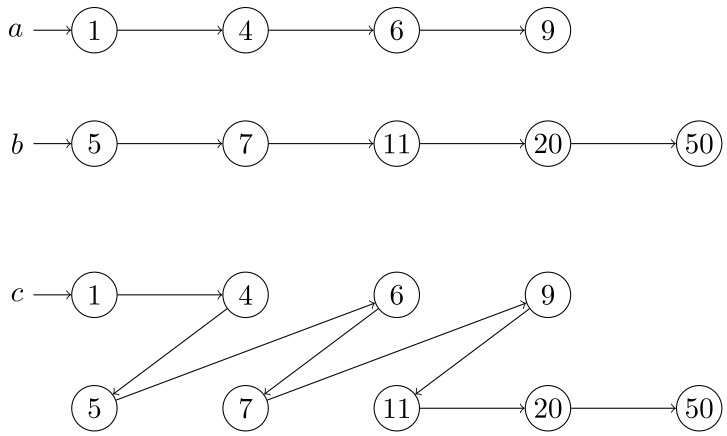
```
List<Comparable> mergeL(List<Comparable> a, List<Comparable> b) {
    if (a.isEmpty()) return b;
    if (b.isEmpty()) return a;

    List<Comparable> c = new List<Comparable>();
    Comparable x = a.read(1);
    Comparable y = b.read(1);
    while (x != null && y != null) {
        if (less(x, y)) {
            c.inserisciInCoda(x);
            a.delete(1);
            x = a.read(1);
        } else {
            c.inserisciInCoda(y);
            b.delete(1);
            y = b.read(1);
        }
    }

    if (x == null) concatena(c, b);
    else concatena(c, a);

    return c;
}
```

3.1 Complessità

Rispetto al merging di vettori, l'implementazione su liste è più efficiente:

- non serve spazio aggiuntivo (se non $O(1)$ per le variabili locali);
- quando una delle due liste si esaurisce, il resto dell'altra viene concatenato in tempo $O(1)$, invece di copiare un elemento alla volta, perciò il tempo di calcolo corrisponde al numero di confronti:
  - $O(1)$ nel caso migliore;
  - $\Theta(n_1 + n_2)$ nel caso peggiore;
  - $O(n_1 + n_2)$ in generale.

3.2 Esempio



4 Divide et impera

La tecnica di progettazione di algoritmi **divide et impera** consiste in due fasi:

1. **divide**: si spezza il problema in sottoproblemi *di ugual dimensione*;
2. **impera**: si risolvono ricorsivamente i sottoproblemi e si uniscono i risultati parziali, formando la soluzione del problema complessivo.

La complessità di un algoritmo divide et impera viene descritta da un'equazione di ricorrenza chiamata, appunto, **equazione divide et impera**,

$$T(n) = mT\left(\frac{n}{a}\right) + f(n)$$

dove:

- $T(n)$ è il costo per risolvere un'istanza del problema di dimensione n;
- m è il numero di sottoproblemi;
- a è il fattore di riduzione della dimensione;
- $f(n)$ è il costo per la scomposizione del problema di dimensione n e la ricomposizione delle soluzioni parziali.

4.1 Soluzione asintotica

Teorema: Nel caso particolare in cui $f(n) = bn^c$, l'equazione divide et impera

$$T(n) = mT\left(\frac{n}{a}\right) + bn^c$$

con $m, a, b, c \in \mathbb{R}^+$, $a > 1$, ha soluzione asintotica

$$T(n) = \begin{cases} \Theta(n^c) & \text{se } m < a^c \\ \Theta(n^c \log n) & \text{se } m = a^c \\ \Theta(n^{\log_a m}) & \text{se } m > a^c \end{cases}$$

4.2 Esempio

La ricerca del minimo e del massimo in un vettore di n elementi si può effettuare con un singolo ciclo, che scandisce gli elementi del vettore uno dopo l'altro, oppure con il metodo divide et impera:

- *divide*: si spezza il vettore in 2 parti uguali;
- *impera*: si risolvono ricorsivamente i 2 sottoproblemi, ottenendo (min_1, max_1) e (min_2, max_2), da cui si ricava la soluzione complessiva

$$(\min(min_1, min_2), \max(max_1, max_2))$$

In questo modo, il numero di confronti eseguiti soddisfa l'equazione

$$T(2) = 1, \quad T(n) = 2T\left(\frac{n}{2}\right) + 2$$

che ha $m = 2, a = 2, b = 2, c = 0$ (perché $f(n) = 2 = 2n^0$), e quindi

$$m > a^c = 1 \implies T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$$

5 Merge sort

L'algoritmo **merge sort** è basato sulla procedura di merging e sul metodo divide et impera:

- *divide*: si spezza la sequenza in due sequenze di ugual lunghezza;
- *impera*: si ordinano ricorsivamente le due sequenze e le si unisce mediante merging.

Esso può essere implementato sia su vettori che su liste. Il codice di un'implementazione ricorsiva (*top-down*) su vettori è:

```
public static void mergesort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int mid = (lo + hi) / 2;
    mergesort(a, lo, mid);
    mergesort(a, mid + 1, hi);
    merge(a, lo, mid, hi);
}
```

Questo è un algoritmo stabile (grazie alla stabilità del merge).

5.1 Complessità

Il numero di confronti eseguiti è determinato dalla procedura `merge`, che in questo caso opera su due sequenze di lunghezza $\frac{n}{2}$:

- il massimo è $n - 1$, quando entrambe le sequenze si esauriscono solo alla fine, e di conseguenza viene effettuato un confronto per ogni dato trasferito (tranne l'ultimo);
- il minimo è $\frac{n}{2}$, se vengono trasferiti prima tutti gli elementi di una sequenza, con un confronto per ciascuno, e poi tutti quelli dell'altra, senza confronti.

Di conseguenza, il numero di confronti $T(n)$ soddisfa

$$2T\left(\frac{n}{2}\right) + \frac{n}{2} \le T(n) \le 2T\left(\frac{n}{2}\right) + n$$

Siccome $m = a = 2$, $c = 1$ (e b varia tra $\frac{1}{2}$ e 1, ma è irrilevante per la soluzione asintotica), si ha $m = a^c = 2$, e quindi

$$T(n) = \Theta(n \log n)$$

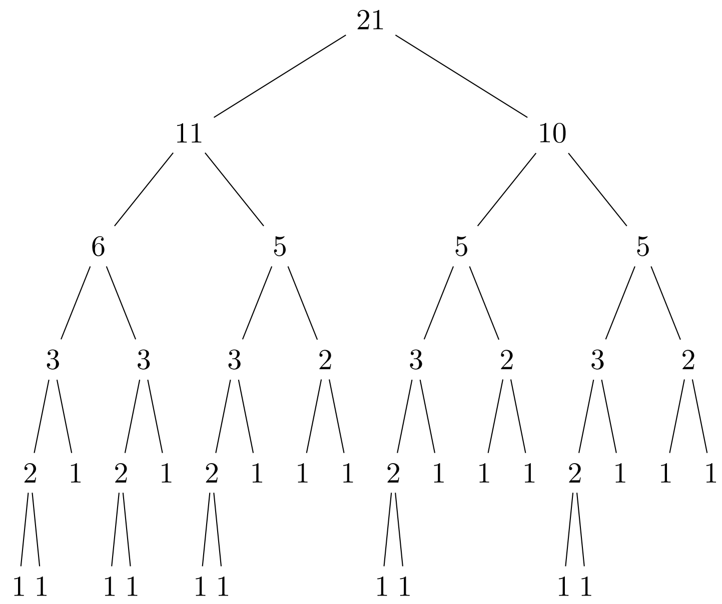
in ogni caso.

Poiché il numero complessivo di operazioni effettuate dal merge su vettori è sempre $\Theta(n)$, lo stesso risultato $\Theta(n \log n)$ vale anche per il tempo di calcolo, e allora l'algoritmo è *ottimale*.

5.2 Albero

Il funzionamento del merge sort top-down può essere rappresentato mediante un albero, i cui nodi riportano le dimensioni dei sottoproblemi.

Ad esempio, l'albero corrispondente a una sequenza di 21 elementi è



Quest'albero è un altro modo per ricavare la complessità dell'algoritmo:

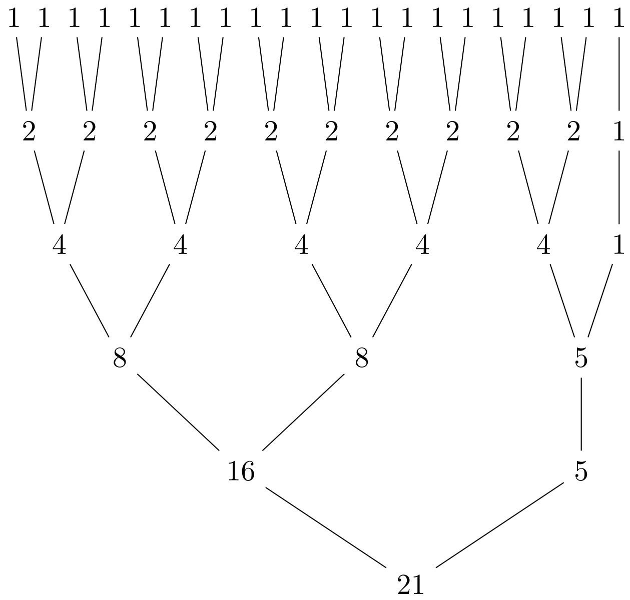
- l'altezza è $\Theta(\log n)$, dato che la dimensione dei sottoproblemi si dimezza a ogni livello;
- il costo totale dei merge a ogni livello è $\Theta(n)$;

quindi la complessità è $\Theta(n \log n)$.

6 Merge sort iterativo

La versione iterativa (*bottom-up*) del merge sort interpreta gli elementi del vettore come sequenze ordinate di lunghezza 1, e a ogni passaggio effettua il merge delle coppie di sequenze adiacenti. Di conseguenza, all'iterazione i crea sequenze ordinate di lunghezza 2^i.

L'albero che ne rappresenta il funzionamento, ad esempio per $n = 21$, è



Esso corrisponde biunivocamente alla rappresentazione binaria di n.

- Se un nodo $n > 1$ è una potenza di 2, i suoi due figli sono entrambi $\frac{n}{2}$;
- Se invece $n > 1$ non è una potenza di 2, lo si può scrivere come $2^k + r$, dove 2^k è il valore del suo bit 1 più significativo. Allora, i suoi due figli sono 2^k e r.

  Ad esempio, $21 = 16 + 5$, quindi il nodo 21 ha figli 16 e 5.

Alcuni nodi hanno un unico figlio, perché nell'algoritmo iterativo può esserci un numero dispari di sequenze, e quindi non è garantito che si possa effettuare il merge dell'ultima con un'altra a ogni passaggio. Ciò non influisce sull'altezza dell'albero.

6.1 Complessità

La complessità in tempo del merge sort iterativo si può ricavare dall'albero.

Nonostante l'albero bottom-up sia diverso da quello top-down (tranne quando n è una potenza di 2), i livelli sono comunque $\Theta(\log n)$, essendo in numero uguale alla lunghezza di n in binario, e il costo totale dei merge su ciascun livello è ancora $\Theta(n)$, quindi il tempo di calcolo è $\Theta(n \log n)$, come per la versione ricorsiva.