3. Algoritmi di Ordinamento MergeSort

Iniziamo ora lo studio di algoritmi di ordinamento più sofisticati e meno intuitivi rispetto a quelli illustrati precedentemente (`selectionSort`, `insertionSort`, `bubbleSort`), ma che risultano più efficienti. Gli algoritmi che considereremo sono `mergeSort`, `quickSort` e `heapSort`. Come vedremo, il primo e il terzo effettuano, nel caso peggiore, un numero di confronti dell'ordine di $n \log n$. Mostreremo anche che, per ordinare n elementi, *ogni algoritmo basato su confronti* deve effettuare, nel caso peggiore, un numero di confronti almeno dell'ordine di $n \log n$. Quindi, dal punto di vista del numero dei confronti, questi due algoritmi non sono migliorabili. L'algoritmo `quickSort` nel caso peggiore effettua un numero di confronti più elevato, ma vedremo che tale caso si presenta raramente.

In questa parte presentiamo il primo dei tre algoritmi.

3.1 Ordinamento per fusione (`mergeSort`)

L'algoritmo di ordinamento per fusione si basa sul seguente schema:

- Un array di un elemento è già ordinato *(caso base)*.
- Per ordinare un array A contenente $n > 1$ elementi possiamo:
  - suddividere A in due array B e C di $n/2$ elementi ciascuno, corrispondenti alla prima e alla seconda metà dell'array A (o di $\lfloor n/2 \rfloor$ e $\lceil n/2 \rceil$ elementi, nel caso n sia dispari),
  - ordinare separatamente gli array B e C,
  - "fondere" gli array ordinati B e C nell'array A, in modo da ottenere un array ordinato contenente gli elementi di B e di C.

L'operazione di fusione (`merge`) di due array ordinati in un array ordinato, che discuteremo poco più avanti, è più semplice rispetto all'ordinamento di un array.

Esempio.
Se l'array A da ordinare contiene inizialmente i numeri

Il presente materiale integra *ma non sostituisce* il libro di testo consigliato.
Data pubblicazione: 20 ottobre 2019

```
7 6 3 8 4 2 10 5
```

Dividiamo l'array in due parti, corrispondenti alla prima e alla seconda metà , ottenendo i seguenti array B e C:

```
7 6 3 8          4 2 10 5
```

Ordinando separatamente i due array B e C otteniamo:

```
3 6 7 8          2 4 5 10
```

Infine, fondendo i due array B e C nell'array A, otteniamo il risultato desiderato:

```
2 3 4 5 6 7 8 10
```

□

Per ordinare i due array B e C utilizziamo *ricorsivamente* lo stesso algoritmo.

Codice 3.1 Schema dell'ordinamento per fusione

Algoritmo *mergeSort (array* $A[0..n-1]$*)*
 if $n > 1$ **then**
  $m \leftarrow n/2$
  $B \leftarrow A[0..m-1]$
  $C \leftarrow A[m..n-1]$
  $mergeSort(B)$
  $mergeSort(C)$
  $A \leftarrow merge(B, C)$

Dallo schema precedente possiamo osservare che per $n \leq 1$ il numero di confronti totali è 0, altrimenti il numero di confronti è dato dal numero di confronti per ordinare l'array B, di lunghezza $\lfloor n/2 \rfloor$, più il numero di confronti per ordinare l'array C, di lunghezza $\lceil n/2 \rceil$, più il numero di confronti per effettuare il merge di B e C, cioè di due vettori che, complessivamente, contengono n elementi. Denotiamo tale numero con $\mathcal{C}_{\texttt{merge}}(n)$. Pertanto, indicando con $\mathcal{C}(n)$ il numero di confronti effettuati da `mergeSort` per ordinare un array di lunghezza n, possiamo scrivere la seguente *equazione di ricorrenza*:

$$\mathcal{C}(n) = \begin{cases} \mathcal{C}(\lfloor n/2 \rfloor) + \mathcal{C}(\lceil n/2 \rceil) + \mathcal{C}_{\texttt{merge}}(n) & \text{if } n > 1 \\ 0 & \text{altrimenti} \end{cases} \tag{3.1}$$

Studiamo ora come effettuare il merge. Disponiamo di due vettori B e C *ordinati* in modo non decrescente, vogliamo ottenere un vettore X che contenga gli stessi elementi di B e C e sia anch'esso ordinato in modo non decrescente. Possiamo risolvere il problema usando il seguente schema:

1. Creiamo un vettore X la cui lunghezza sia la somma delle lunghezze di B e di C, inizialmente vuoto.
2. Ispezioniamo B e C iniziando a considerare gli elementi minimi, cioè gli elementi che si trovano nella prima posizione in ciascuno dei due vettori.
3. Confrontiamo i due elementi che stiamo considerando in B e C e scegliamo il minimo, copiandolo nella prima posizione libera del vettore X. Inoltre, nel vettore da cui abbiamo scelto il minimo, passiamo a considerare l'elemento successivo a quello copiato.
4. Ripetiamo le operazioni precedenti, fino a raggiungere la fine di uno dei due array.
5. Copiamo in X tutti gli elementi non ancora utilizzati dell'altro array.

Codice 3.2 Fusione (merge) di due array ordinati

```
Algoritmo merge (array B[0..ℓ_B − 1], array C[0..ℓ_C − 1]) → array
  Sia X[0..ℓ_B + ℓ_C − 1] un array
  i_1 ← 0                                  // prossima posizione da considerare in B
  i_2 ← 0                                  // prossima posizione da considerare in C
  k ← 0                                      // prossima posizione da riempire in X
  while i_1 < ℓ_B and i_2 < ℓ_C do  // non raggiunta né la fine di B né quella di C
      if B[i_1] ≤ C[i_2] then
          X[k] ← B[i_1]                        // preleva il prossimo elemento da B
          i_1 ← i_1 + 1
      else
          X[k] ← C[i_2]                        // preleva il prossimo elemento da C
          i_2 ← i_2 + 1
      k ← k + 1
  if i_1 < ℓ_B then
      for j ← i_1 to ℓ_B − 1 do          // copia in X gli elementi rimasti in B
          X[k] ← B[j]
          k ← k + 1
  else
      for j ← i_2 to ℓ_C − 1 do          // copia in X gli elementi rimasti in C
          X[k] ← C[j]
          k ← k + 1
  return X
```

Numero di confronti di `merge`

Calcoliamo ora il numero totale di confronti $\mathcal{C}_{\texttt{merge}}(n)$ tra chiavi effettuati dall'algoritmo `merge`, nel caso che il numero totale di elementi dei due vettori da fondere sia n (cioè $\ell_B + \ell_C = n$, con riferimento al Codice 3.2). Osserviamo che dopo ciascun confronto ($B[i_1] \leq C[i_2]$, nel Codice 3.2), viene sistemato un elemento nell'array X. Poiché il numero totale di elementi da sistemare è n, il numero di confronti è limitato da n. In realtà possiamo ridurre tale limite a $n-1$. Infatti, a un certo punto dell'esecuzione verrà raggiunta la fine di uno dei due array, mentre nell'altro vi sarà almeno un elemento che sarà copiato in X senza effettuare alcun confronto. Per esercizio, costruite degli esempi in cui siano *necessari* $n-1$ confronti. Possiamo dunque concludere che, nel caso peggiore, $\mathcal{C}_{\texttt{merge}}(n) = n-1$. Per esercizio, trovate anche il numero di confronti minimo, cioè corrispondente al caso migliore.

Numero di confronti di `mergeSort`

Siamo ora in grado di calcolare il numero di confronti effettuati da `mergeSort`, nel caso peggiore, sostituendo nell'equazione (3.1) il valore appena calcolato per $\mathcal{C}_{\texttt{merge}}(n)$:

$$\mathcal{C}(n) = \begin{cases} \mathcal{C}(\lfloor n/2 \rfloor) + \mathcal{C}(\lceil n/2 \rceil) + n - 1 & \text{se } n > 1 \\ 0 & \text{altrimenti} \end{cases} \tag{3.2}$$

Supponendo n pari, otteniamo:

$$\mathcal{C}(n) = \begin{cases} 2\mathcal{C}(n/2) + n - 1 & \text{se } n > 1 \\ 0 & \text{altrimenti} \end{cases} \tag{3.3}$$

Risolvendo per sostituzione, nel caso di n potenza di 2 otteniamo $\mathcal{C}(n) = n \log_2 n - n + 1 = \Theta(n \log n)$. Se n non è una potenza di due, osserviamo che $\mathcal{C}(n) \leq \mathcal{C}(N) = \Theta(N \log N)$ dove N è

la più piccola potenza di 2 maggiore di n. Da risultati di teoria dei numeri, è noto che per ogni intero n, la più piccola potenza di 2 maggiore o uguale a n è minore di $2n$. Dunque $N < 2n$, che implica $\Theta(N \log N) = \Theta(n \log n)$. Questo permette di concludere che $\mathcal{C}(n) = \Theta(n \log n)$, anche nel caso di n non potenza di 2.

Tempo di calcolo

Dal calcolo del numero di confronti non possiamo ottenere immediatamente una stima del tempo di calcolo dell'algoritmo. Infatti, implementando lo schema dell'algoritmo presentato nel Codice 3.1 in modo diretto, vi sono varie operazioni costose sia in termini di tempo che in termini di spazio: dobbiamo creare due array B e C, copiarvi gli elementi di A e, dopo il merge, ricopiare tutti gli elementi dell'array ottenuto nell'array iniziale. Queste operazioni utilizzano una certa quantità di tempo e spazio e vengono effettuate ogni volta che l'algoritmo viene richiamato su un array di lunghezza maggiore di 1. Indicando con $T(n)$ il tempo utilizzato dall'algoritmo per ordinare un array di lunghezza n, osserviamo che:

- Se l'array contiene al più un elemento, l'algoritmo utilizza tempo costante. Indichiamo tale tempo con a.
- Se l'array contiene $n > 1$ elementi, allora $T(n)$ è la somma dei seguenti tempi:
  - Tempo per la creazione degli array B e C e la copia in essi degli elementi di A. Tale tempo è $\Theta(n)$.
  - Tempo per ordinare B e C, cioè $T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$
  - Tempo per il merge che, come discusso, utiilizza al più $n-1$ confronti. *Nell'ipotesi che ogni confronto sia effettuato in tempo costante*, il tempo per il merge è $\Theta(n)$.

  Dunque $T(n)$ è della forma $\Theta(n)+T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+\Theta(n) = T(\lfloor n/2 \rfloor)+T(\lceil n/2 \rceil)+\Theta(n)$. Possiamo scrivere l'ultimo termine della somma, cioè $\Theta(n)$, come $bn + c$, dove b e c sono due costanti. Pertanto $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + bn + c$.

Per n pari, otteniamo la seguente equazione di ricorrenza:

$$T(n) = \begin{cases} 2T(n/2) + bn + c & \text{se } n > 1 \\ a & \text{altrimenti} \end{cases} \tag{3.4}$$

Risolvendo per sostituzione, nel caso di n potenza di 2 si ottiene $T(n) = an + bn \log_2 n + c(n-1)$, da cui concludiamo $T(n) = \Theta(n \log n)$, avendo supposto che ogni confronto usi tempo costante.

Implementazione di `mergeSort`

Come osservato, l'implementazione diretta dello schema presentato nel Codice 3.1 richiede l'uso di tempo e spazio per gli array B e C. Possiamo implementare l'algoritmo in maniera differente (v. Codice 3.3), evitando l'uso degli array B e C e servendoci direttamente dell'array A stesso e di due indici, che delimitano la parte da ordinare. Al contrario, per effettuare il merge utilizzeremo un array ausiliario. Tuttavia, per evitare inutile spreco di memoria, possiamo creare preliminarmente un *unico* array ausiliario utilizzato da tutte le chiamate di `merge`.[1]

La precedente analisi relativa al numero di confronti non cambia. Infatti nel Codice 3.3 l'uso delle operazioni di confronto non è cambiato rispetto al Codice 3.1 e al Codice 3.2 analizzati in precedenza. Nel caso ciascuna operazione di confronto costi tempo costante, con un'analisi simile alla precedente, possiamo mostrate che il tempo è dell'ordine di $n \log n$.

[1] È possibile effettuare il merge, *senza utilizzare array ausiliari*, e con un *numero di confronti lineare*, come nell'algoritmo che abbiamo presentato. I pochi algoritmi noti per fare ciò sono estremamente complicati (si veda ad esempio l'articolo: V. Geffert, J. Katajainen, T. Pasanen, Asymptotically efficient in-place merging, *Theor. Comput. Sci.* 237(1-2): 159-181 (2000)).

Codice 3.3 Ordinamento per fusione

Al posto degli array ausiliari B e C (che richiedono uso di memoria e di tempo per il trasferimento dei dati) si utilizza l'array A stesso con indici che delimitano le parti da ordinare. Si utilizza un array ausiliario X per le operazioni di merge. X potrebbe essere definito localmente alla procedura *merge*, in quanto viene utilizzato solo da essa. In questo modo tuttavia occorrerebbe un nuovo array ausiliario ad ogni chiamata di *merge*. Per evitare ciò, l'array ausiliario viene definito a livello globale.

```
Algoritmo mergeSort (array A[0..n − 1])
  Sia X[0..n − 1] un array
  mergeSort(A, 0, n, X)

Procedura mergeSort (array A, indice i, indice f, array X)
                          /* Ordina A[i..f − 1] utilizzando X come array ausiliario */
  if f − i > 1 then
      m ← (i + f)/2
      mergeSort(A, i, m, X)
      mergeSort(A, m, f, X)
      merge(A, i, m, f, X)

Procedura merge (array A, indice i, indice m, indice f, array X)
    /* Merge tra A[i..m − 1] e A[m..f − 1] utilizzando X come array ausiliario */

  i1 ← i          // Prima parte: merge di A[i..m − 1] e A[m..f − 1] in X[0..f − i − 1]
  i2 ← m
  k ← 0
  while i1 < m and i2 < f do
      if A[i1] ≤ A[i2] then
          X[k] ← A[i1]
          i1 ← i1 + 1
      else
          X[k] ← A[i2]
          i2 ← i2 + 1
      k ← k + 1
  if i1 < m then
      for j ← i1 to m − 1 do
          X[k] ← A[j]
          k ← k + 1
  else
      for j ← i2 to f − 1 do
          X[k] ← A[j]
          k ← k + 1
  for k ← 0 to f − i − 1 do      // Seconda parte: copia il risultato in A[i..f − 1]
      A[i + k] ← X[k]
```

Spazio

Studiamo ora lo spazio utilizzato dal `mergeSort` (Codice 3.3). Prima di tutto, osserviamo che, oltre all'array da ordinare, l'algoritmo non è *in loco*, in quanto utilizza un array di supporto per effettuare il merge. L'array è di n elementi e, dunque, usa spazio $\Theta(n)$ (nell'ipotesi che ogni cella dell'array occupi spazio $\Theta(1)$, ad esempio se si tratta di un array di interi o di puntatori). Dobbiamo inoltre considerare lo spazio utilizzato sullo stack per gestire la ricorsione. A tale scopo, osserviamo che in ciascun record di attivazione di `mergeSort` devono essere memorizzati gli indici i e f che delimitano la porzione di array da ordinare, e la variabile m (gli array A e X sono invece fissati). Pertanto la dimensione di ogni record di attivazione è costante.

Calcoliamo ora l'altezza dello stack, cioè il numero massimo di record di attivazione $\mathcal{H}(n)$ presenti sullo stack, nel caso si debba ordinare un array contenente n elementi. Come nel caso dei confronti possiamo considerare la struttura ricorsiva dell'algoritmo e scrivere un'equazione di ricorrenza.

- Se $n \leq 1$ (caso base) non viene effettuata alcuna chiamata ricorsiva. Pertanto viene utilizzato solo il record di attivazione corrente. Dunque $\mathcal{H}(n) = 1$.
- Se $n > 1$ viene effettuata una prima chiamata ricorsiva, su un array di lunghezza $\lfloor n/2 \rfloor$, che dunque utilizzerà altezza $\mathcal{H}(\lfloor n/2 \rfloor)$. Terminata tale chiamata, si effettuata una seconda chiamata, su un array di lunghezza $\lceil n/2 \rceil$, che utilizza altezza $\mathcal{H}(\lceil n/2 \rceil)$. Poiché al termine di ciascuna chiamata ricorsiva lo stack viene riportato all'altezza che aveva prima della chiamata, la parte di stack utilizzata dalla prima chiamata *viene riutilizzata* per la seconda. Pertanto l'altezza dello stack utilizzata dalle due chiamate è il massimo tra $\mathcal{H}(\lfloor n/2 \rfloor)$ e $\mathcal{H}(\lceil n/2 \rceil)$, a cui va aggiunto il record di attivazione corrente.

Otteniamo dunque:

$$\mathcal{H}(n) = \begin{cases} \max\left(\mathcal{H}(\lfloor n/2 \rfloor), \mathcal{H}(\lceil n/2 \rceil)\right) + 1 & \text{se } n > 1 \\ 1 & \text{altrimenti} \end{cases} \tag{3.5}$$

che nel caso di n pari possiamo semplificare in:

$$\mathcal{H}(n) = \begin{cases} \mathcal{H}(n/2) + 1 & \text{se } n > 1 \\ 1 & \text{altrimenti} \end{cases} \tag{3.6}$$

che ha soluzione $\mathcal{H}(n) = 1 + \log_2 n$, per n potenza di 2. Questo ci permette di concludere che l'altezza dello stack è logaritmica rispetto a n. Inoltre poiché ogni record di attivazione ha una dimensione costante, cioè non dipendente da n, la quantità di stack utilizzata dall'algoritmo è $\Theta(\log n)$.