Capitolo 7

Algoritmi di ordinamento

7.1 Selection sort

L'algoritmo di ordinamento per selezione opera nel modo seguente: supponiamo che i primi k elementi siano ordinati; l'algoritmo sceglie il minimo degli $n - k$ elementi non ancora ordinati e lo inserisce in posizione $k + 1$; partendo da $k = 0$ e iterando n volte il ragionamento, si ottiene la sequenza interamente ordinata.

algoritmo `selectionSort` (*vettore* A) $\rightarrow$ *void*

```
1 for k = 0 to length(A) - 2 do
2   m = k + 1
3   for j = m + 1 to length(A) do
4     if(A[j] < A[m]) then
5       m = j
6   swap(A[m], A[k + 1])
```

Complessità

L'estrazione del minimo (righe 2 - 4) richiede $n - k - 1$ confronti e, poiché il ciclo esterno viene eseguito $n - 1$ volte, si ha:

$$T(n) = \sum_{k=0}^{n-2}(n - k - 1) = \sum_{i=1}^{n-1} i = \Theta(n^2)$$

7.2 Insertion sort

L'algoritmo di ordinamento per inserzione opera in modo simile al precedente: supponiamo che i primi k elementi siano ordinati; l'algoritmo prende il $(k+1)$-esimo elemento e lo inserisce nella posizione corretta rispetto ai k elementi già ordinati; partendo da $k = 0$ ed iterando n volte il ragionamento, il vettore viene completamente ordinato.

algoritmo `insertionSort` (*vettore* A) $\rightarrow$ *void*

```
1 for k = 1 to length(A) - 1 do
2     value = A[k + 1]
3     j = k
4     while(j > 0 && A[j] > value) do
5         A[j + 1] = A[j]
6         j--
7     A[j + 1] = value
```

Complessità

Le righe 4 - 6 individuano la posizione in cui l'elemento va inserito, eseguendo al più k confronti; poiché il ciclo esterno viene eseguito $n-1$ volte, si ha:

$$T(n) = \sum_{k=1}^{n-1} k = O(n^2)$$

7.3 Bubble sort

L'algoritmo di ordinamento a bolle opera una serie di scansioni del vettore: in ogni scansione sono confrontate coppie di elementi adiacenti e viene effettuato uno scambio se i due elementi non rispettano l'ordinamento; se durante una scansione non viene eseguito alcuno scambio, allora il vettore è ordinato.

algoritmo `bubbleSort` (*vettore* A) $\rightarrow$ *void*

```
1 for i = 1 to length(A) - 1 do
2   for j = 2 to (n - i + 1) do
3     if(A[j - 1] > A[j])
4       swap(A[j - 1], A[j])
5     if(non ci sono stati scambi)
6       break
```

Lemma 7.1. *Dopo l'i-esima scansione, gli elementi $A[n-i+1], ..., A[n]$ sono correttamente ordinati e occupano la loro posizione definitiva, ovvero $\forall h, k$ con $1 \leq h \leq k \wedge n-i+1 \leq k \leq n$, risulta $A[h] \leq A[k]$.*

Dimostrazione. Procediamo per induzione sul numero i dell'iterazione.

Caso base: si ha $i = 1$; dopo la prima scansione, l'elemento massimo di A ha raggiunto l'n-esima posizione, quindi il passo base è facilmente verificato.

Ipotesi induttiva: supponiamo che l'enunciato valga per le prime $i-1$ scansioni.

Passo induttivo: verifichiamo la validità della proprietà dopo l'i-esima scansione; al termine di tale scansione, il massimo degli elementi $A[1], ..., A[n-i+1]$ avrà raggiunto la posizione $n-i+1$ e quindi, per ogni $h < n-i+1$, risulta $A[h] \leq A[n-i+1]$; combinando questa osservazione con l'ipotesi induttiva, si può verificare facilmente la validità dell'enunciato.

Complessità

Per il lemma appena dimostrato, dopo al più $n-1$ cicli il vettore è ordinato e, nel caso peggiore, all'i-esima scansione vengono eseguiti $n-i$ confronti; si ottiene quindi:

$$T(n) = \sum_{i=1}^{n-1}(n-i) = \sum_{j=1}^{n-1} j = O(n^2)$$

7.4 Heap sort

L'algoritmo di ordinamento heap sort, come dice il nome, si basa sull'utilizzo di una struttura dati di tipo heap.

algoritmo `heapSort` (*vettore* A) $\rightarrow$ *void*

```
1 buildHeap(A)
2 for i = length(A) - 1 downto 1 do
3   swap(A[i], A[1])
4   heapifyDown(A[1..(i - 1)], 1)
```

Complessità

La funzione `buildHeap` ha complessità $O(n)$, mentre `heapifyDown`, che viene richiamata n volte, ha complessità $O(\log n)$; in totale si ha:

$$T(n) = O(n) + n \cdot O(\log n) = O(n \cdot \log n)$$

7.5 Merge sort

L'algoritmo di ordinamento per fusione si basa sulla tecnica del *divide et impera*:

Divide: si divide il vettore in due parti di uguale dimensione; se ha un numero dispari di elementi, una delle due parti conterrà un elemento in più.

Impera: supponendo di avere le due sottosequenze ordinate, si fondono in una sola sequenza ordinata.

algoritmo `mergeSort` (*vettore A, indice i, indice f*) → *void*

```
1  if(i < f)
2    m = (i + f) / 2
3    mergeSort(A, i, m)
4    mergeSort(A, m + 1, f)
5    merge(A, i, m, f)

6  merge(vettore A, indice i1, indice f1, indice f2)
7  X: vettore[f2 - i1 + 1]
8  i = 1
9  i2 = f1 + 1
10 while(i1 < f1 && i2 < f2) do
11   if(A[i1] <= A[i2]) then
12     X[i] = A[i1]
13     i++, i1++
14   else
15     X[i] = A[i2]
16     i++, i2++
17 if(i1 < f1)
18   copia A[i1..f1] alla fine di X
19 else
20   copia A[i2..f2] alla fine di X
21 copia X in A[i1..f2]
```

Complessità

La complessità della funzione ausiliaria `merge` è $O(n)$; la relazione di ricorrenza è la seguente:

$$T(n) = 2 \cdot T(n/2) + O(n)$$

Applicando il secondo caso del teorema fondamentale, si ottiene:

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

Per quanto riguarda lo spazio di memoria, `mergeSort` non opera in loco e pertanto ha un'occupazione di memoria pari a $2 \cdot n$.

7.6 Quicksort

Anche questo tipo di algoritmo segue un approccio di tipo *divide et impera*:

Divide: sceglie un elemento x della sequenza (il *perno*), partiziona la sequenza in due sottosequenze contenenti, rispettivamente, gli elementi minori o uguali a x e gli elementi maggiori di x, e ordina ricorsivamente le due sottosequenze.

Impera: concatena le sottosequenze ordinate.

algoritmo `quickSort` (*vettore A, indice i, indice f*) → *void*

```
1  if(i < f)
2    m = partition(A, i, f)
3    quickSort(A, i, m - 1)
4    quickSort(A, m + 1, f)

5  partition(vettore A, indice i, indice f) → indice
6  x = A[i]
7  inf = i
8  sup = f + 1
9  while(true) do
10   do inf++ while(A[inf] <= x)
11   do sup-- while(A[sup] > x)
12   if(inf < sup) then
13     swap(A[inf], A[sup])
14   else
15     break
16 swap(A[i], A[sup])
17 return sup
```

L'invariante della procedura `partition` è il seguente:

Proprietà 7.1. In ogni istante, gli elementi $A[i], ..., A[inf - 1]$ sono minori o uguali al perno, mentre gli elementi $A[sup + 1], ..., A[f]$ sono maggiori del perno.

Complessità

L'algoritmo ordina un vettore A di dimensione n suddividendolo in due sottovettori, A_1 e A_2, di dimensioni a e b tali che $a + b = n - 1$, ed ordinando ricorsivamente A_1 e A_2; poiché la partizione richiede $n - 1$ confronti, si può scrivere la seguente relazione di ricorrenza:

$$T(n) = T(a) + T(b) + n - 1$$

Nel caso peggiore, si ha che ad ogni passo viene scelto come perno l'elemento più piccolo (o più grande) della porzione di vettore considerato, riducendo la relazione di ricorrenza a:

$$T(n) = T(n-1) + n - 1 = O(n^2)$$

Dal risultato sembra che `quickSort` sia un algoritmo inefficiente; l'idea per migliorarlo consiste nell'usare la *randomizzazione*. Supponiamo di scegliere il perno come il k-esimo elemento di A, dove k è *scelto a caso*: per calcolare il numero atteso di confronti, bisogna calcolare la somma dei tempi di esecuzione pesandoli in base alla probabilità di fare una certa scelta casuale; in questo caso, la scelta è il valore di k e la probabilità di scegliere il k-esimo elemento come perno, $\forall k, 1 \leq k \leq n$ è $1/n$. Otteniamo la seguente relazione di ricorrenza:

$$T(n) = \sum_{a=0}^{n-1} \frac{1}{n} \cdot (T(a) + T(n-a-1) + n - 1) = n - 1 + \sum_{a=0}^{n-1} \frac{2}{n} \cdot T(a) \tag{7.1}$$

Dimostriamo, usando il metodo di sostituzione, che $T(n) = O(n \cdot \log n)$, ossia che esiste una costante $\alpha > 0$ tale per cui $C(n) \leq \alpha \cdot n \cdot \log n$ per ogni $n \geq n_0, n_0 > 0$.

Passo base: si ha $n = 1$; $T(1) = 0 = \alpha \cdot (1 \cdot \log 1)$ per ogni α.

Ipotesi induttiva: supponiamo che esista α tale per cui $T(i) \leq \alpha \cdot i \cdot \log i$ per $i < n$.

Passo induttivo: usando l'ipotesi induttiva nella relazione 7.1 e la definizione di somme integrali, si ottiene:

$$T(n) = n-1+\sum_{i=0}^{n-1} \frac{2}{n} \cdot T(i) \leq n-1+\sum_{i=0}^{n-1} \frac{2}{n} \cdot \alpha \cdot i \cdot \log i = n-1+\frac{2 \cdot \alpha}{n} \cdot \sum_{i=2}^{n-1} i \cdot \log i \leq n-1+\frac{2 \cdot \alpha}{n} \cdot \int_2^n x \cdot \log x dx$$

Procediamo usando il metodo di integrazione per parti:

$$T(n) \leq n-1+\frac{2\cdot\alpha}{n}\cdot\left(n^2\cdot\frac{\log n}{2}-\frac{n^2}{4}+2\cdot\ln 2+1\right)=n-1+\alpha\cdot n\cdot\log n-\alpha\cdot\frac{n}{2}-O(1)\leq\alpha\cdot n\cdot\log n$$

quando $n-1<\alpha\cdot(n/2)$, ossia per $\alpha\geq 2$.

Possiamo dunque concludere che l'algoritmo `quickSort`, su un input di dimensione n ha complessità $O(n^2)$ nel caso peggiore, ma il numero atteso di confronti è $O(n\cdot\log n)$.

7.7 Counting sort

Si tratta di un algoritmo per ordinare vettori di interi di n elementi compresi nell'intervallo [1, k].

algoritmo `countingSort` (*vettore A, intero k*) → *void*

```
1  X: vettore[k]
2  for i = 1 to k do
3    X[i] = 0
4  for i = 1 to n do
5    X[A[i]]++
6  j = 1
7  for i = 1 to n do
8    while(X[i] > 0) do
9      A[j] = i
10     j++, X[i]--
```

Complessità

Il primo ciclo ha complessità $O(k)$, il secondo $O(n)$ e il terzo, nonostante sia costituito da due cicli innestati, è anch'esso $O(n)$; per rendersene conto, basta osservare che il tempo necessario per eseguire gli n incrementi dev'essere uguale a quello necessario per gli n decrementi, e gli incrementi sono eseguiti dal secondo ciclo. In totale la complessità risulta essere:

$$T(n)=O(n+k)$$

L'algoritmo `countingSort` è assai efficiente se $k=O(n)$, mentre è preferibile usarne altri all'aumentare dell'ordine di grandezza di k; si osservi, inoltre, che la memoria occupata è anch'essa $O(n+k)$.

7.8 Radix sort

Si tratta di un algoritmo per ordinare interi in tempo lineare, quando l'intervallo dei numeri che possono essere presenti nel vettore è troppo grande per pensare di utilizzare il `countingSort`; consiste nell'ordinare, ad ogni ciclo, in base all'i-esima cifra, partendo da quella meno significativa fino a quella più significativa.

algoritmo `radixSort` (*vettore A*) → *void*

```
1  t = 0
2  while(esiste un numero con la t-esima cifra diversa da 0)
3    bucketSort(A, 10, t)
4    t++

5  bucketSort(vettore A, intero b, intero t)
6  Y: vettore[b]
7  for i = 1 to b do
8    Y[i] = lista vuota
9  for i = 1 to n do
10   c = t-esima cifra di A[i] in base b
11   attacca A[i] alla lista Y[c + 1]
12 for i = 1 to b do
13   copia ordinatamente in A gli elementi di Y[i]
```

Complessità

Sia k il valore più grande presente nel vettore; l'algoritmo esegue $\log_{10} k$ chiamate di `bucketSort`, ciascuna delle quali ha costo $O(n)$, per un totale di $O(n \cdot \log k)$; è possibile aumentare il valore della base usata nel `bucketSort`, senza però aumentare significativamente il tempo di esecuzione, come afferma il seguente teorema.

Teorema 7.1. *Usando come base per il* `bucketSort` *un valore $b = \Theta(n)$, l'algoritmo* `radixSort` *ordina n interi in $[1, k]$ in tempo*

$$O\left(n \cdot \left(1 + \frac{\log k}{\log n}\right)\right)$$

Dimostrazione. Ci sono $\log_b k = O(\log_n k)$ passate di `bucketSort`, ciascuna delle quali dal costo $O(n)$: il tempo totale è $O(n \cdot \log_n k) = O(n \cdot \log k / \log n)$, per le regole del cambiamento di base dei logaritmi; per considerare il caso $k < n$, aggiungiamo $O(n)$ alla complessità, che è il tempo richiesto per leggere la sequenza. Il risultato della somma è quanto si voleva dimostrare. □

7.9 Limitazione inferiore per algoritmi basati sul confronto

Usiamo la seguente proprietà per stabilire la limitazione:

Proprietà 7.2. Il numero di confronti fatti dall'algoritmo A nel caso peggiore è pari all'altezza dell'albero di decisione, ovvero alla lunghezza del più lungo cammino dalla radice ad una foglia.

Lemma 7.2. *Un albero di decisione per l'ordinamento di n elementi contiene almeno $n!$ foglie.*

Dimostrazione. Ad ogni foglia dell'albero di decisione non ci sono più confronti da effettuare, dunque corrisponde ad una soluzione del problema dell'ordinamento; ogni soluzione corrisponde, a sua volta, ad una permutazione degli n elementi da ordinare, quindi l'albero deve contenere un numero di foglie pari almeno al numero di permutazioni degli n elementi da ordinare, ossia $n!$ foglie. □

Lemma 7.3. *Sia T un albero binario in cui ogni nodo interno ha esattamente 2 figli e sia k il numero delle sue foglie: l'altezza è almeno $\log_2 k$.*

Dimostrazione. Definiamo $h(k)$ come l'altezza di un albero binario con k foglie (con 2 figli per ogni nodo interno) e dimostriamo il lemma per induzione su k.

Caso base: si ha $k = 1$; si tratta, dunque, di un albero con un solo nodo, quindi l'altezza è $0 \geq \log_2 1$.

Ipotesi induttiva: assumiamo che $h(i) \geq \log_2 i$ per $i < k$.

Passo induttivo: poiché uno dei sottoalberi ha almeno la metà delle foglie, si ha:

$$h(k) \geq 1 + h\left(\frac{k}{2}\right) \geq 1 + \log_2\left(\frac{k}{2}\right) = 1 + \log_2 k - \log_2 2 = \log_2 k$$

come volevasi dimostrare. □

Teorema 7.2. *Il numero di confronti necessari per ordinare n elementi nel caso peggiore è $\Omega(n \cdot \log n)$.*

Dimostrazione. Per il lemma 7.2, il numero di foglie di un albero di decisione è almeno $n!$ e, per la proprietà 7.2, il numero di confronti necessari per l'ordinamento è, nel caso peggiore, pari all'altezza dell'albero, ossia $\geq \log n!$ per il lemma 7.3. La disuguaglianza di De Moivre - Stirling afferma che, per n sufficientemente grande, si ha:

$$\sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n \cdot \left(1 + \frac{1}{12 \cdot n - 1}\right)$$

Si ha dunque:

$$n! \approx \sqrt{2 \cdot \pi \cdot n} \cdot \left(\frac{n}{e}\right)^n$$

da cui:

$$\log n! \approx n \cdot \log n - 1.4427 \cdot n - \frac{1}{2} \cdot \log n + 0.826$$

Da questa relazione segue direttamente la limitazione inferiore che si voleva dimostrare. □