ESERCIZIO DI ASD DEL 17 NOVEMBRE 2008

Pochi da Ordinare

Algoritmo di Ordinamento Stabile. Analizzando gli algoritmi di ordinamento che conosciamo ci accorgiamo che abbiamo almeno due possibilità:

- Modificare SelectionSort utilizzando un vettore ausiliario;
- Modificare CountingSort per fare in modo che operi su k interi distinti, anche se molto grandi.

Proponiamo la soluzione per entrambe le alternative.

SelectionSort ricerca ad ogni iterazione il minimo tra gli elementi rimasti da considerare e lo colloca nella posizione esatta. In generale ha complessità $\Theta(n^2)$. Tuttavia possiamo ottimizzarlo per operare su vettori con k elementi distinti nel seguente modo:

- Ricerco il minimo per k volte (ogni volta solo tra gli elementi rimasti). Per fare questo mi basta una scansione del vettore.
- Una volta trovato il minimo eseguo un'altra scansione del vettore e copio tutte le occorrenze del minimo in un vettore ausiliario che conterrà il risultato finale.

Per essere sicuri di non considerare più volte gli stessi elementi, ogni volta che un numero viene copiato in B il suo valore in A viene modificato assegnandogli un valore maggiore del massimo di A.

Algorithm 1 RiSelectionSort(A, k)

```
1: max ←TrovaMax(A)
2: h ← 1
3: for i ← 1 to k do
4:    min ← TrovaMin(A)
5:    for j ← 1 to length[A] do
6:       if A[j] = min then
7:          B[h] ← A[j]
8:          h ← h + 1
9:          A[j] ← max + 1
10:      end if
11:   end for
12: end for
```

Passiamo alla seconda soluzione, che consiste nel modificare CountingSort. Il vettore A contiene relativamente pochi elementi distinti, in quanto k è una costante e non aumenta all'aumentare della dimensione del vettore. Purtroppo non possiamo applicare immediatamente CountingSort perché gli elementi di A potrebbero essere molto grandi. Per esempio A potrebbe contenere k elementi distinti

Date: 17 Novembre 2008.

Algorithm 2 TrovaMax(A)

```
1: max ← A[1]
2: for i ← 2 to length[A] do
3:    if A[i] > max then
4:       max ← A[i]
5:    end if
6: end for
7: return max
```

Algorithm 3 TrovaMin(A)

```
1: min ← A[1]
2: for i ← 2 to length[A] do
3:    if A[i] < min then
4:       min ← A[i]
5:    end if
6: end for
7: return min
```

compresi tra n^{100} e n^{1000}. Tuttavia possiamo prendere spunto da CountingSort per disegnare un algoritmo di ordinamento stabile e lineare adatto al problema proposto.

Utilizziamo un vettore C di lunghezza k i cui elementi hanno due campi: $C[i].key$ che conterrà uno dei k elementi che compaiono in A e $C[i].occ$ che ci dirà quante occorrenze di $C[i].key$ si trovano in A. Con una scansione del vettore A possiamo riempire il vettore C, esattamente come si fa in CountingSort, con l'unica differenza che per ogni elemento di A che consideriamo dobbiamo trovare la sua "posizione" in C.

Una volta riempito il vettore C possiamo ordinarlo rispetto al campo key. Questo ci costerà poco, indipendentemente dall'algoritmo di ordinamento che useremo perché C ha dimensione k.

Ora possiamo procedere come in CountingSort, "sommando" gli elementi di C e poi scrivendo il risultato finale in B.

Algoritmo di Ordinamento in Place. Cerchiamo di sistemare una delle due soluzioni prima proposte per ottenere un algoritmo in place, con il rischio di perdere la stabilità. SelectionSort usa solo un vettore ausiliario B per il risultato, quindi sembra più facile da adattare. Procediamo nel seguente modo:

- manteniamo un indice j che ci dirà quale è la porzione di A ancora da considerare;
- eseguiamo come prima la ricerca del minimo per k volte, ma ora cercheremo il minimo in $A[j..length[A]]$;
- scandiamo nuovamente $A[j..length[A]]$ per portare ogni occorrenza del minimo nella posizione corretta eseguendo degli scambi.

Quindi all'interno della i-esima iterazione di ricerca e sistemazione dei minimi ci servirà un altro indice pos che ci dirà dove posizionare le occorrenze del minimo. In particolare pos verrà inizializzato all'inizio del ciclo con valore j ed al termine

Algorithm 4 RiCountingSort(A, k)

```
1: for i ← 1 to k do
2:    C[i].key ← −1
3:    C[i].occ ← 0
4: end for
5: for j ← 1 to length[A] do
6:    i ← 1
7:    while C[i].key ≠ −1 and C[i].key ≠ A[j] do
8:       i ← i + 1
9:    end while
10:   C[i].key ← A[j]
11:   C[i].occ ← C[i].occ + 1
12: end for
13: InsertionSort(C)
14: for i ← 2 to k do
15:   C[i].occ ← C[i − 1].occ + C[i].occ
16: end for
17: for j ← length[A] down to 1 do
18:   i ← 1
19:   while C[i].key ≠ A[j] do
20:      i ← i + 1
21:   end while
22:   B[C[i].occ] ← A[j]
23:   C[i].occ ← C[i].occ − 1
24: end for
```

del ciclo verrà utilizzato per aggiornare j. È possibile evitare l'utilizzo di questo indice aggiuntivo sostituendo il ciclo for con un while.

Algorithm 5 RiRiSelectionSort(A, k)

```
1: j ← 1
2: for i ← 1 to k do
3:    pos ← j
4:    min ←RiTrovaMin(A, j, length[A])
5:    for h ← j to length[A] do
6:       if A[h] = min then
7:          Scambia(A, h, pos)
8:          pos ← pos + 1
9:       end if
10:   end for
11:   j ← pos
12: end for
```

Si noti che RiRiSelectionSort non è stabile. Trovate un vettore che dimostri questa affermazione.

Complessità. Tutti gli algoritmi presentati hanno complessità $\Theta(n)$. Infatti al massimo troviamo due cicli for innestati, di cui uno con indici che variano da 1 ad

Algorithm 6 RiTrovaMin(A, r, s)

```
1: min ← A[r]
2: for i ← r + 1 to s do
3:    if A[i] < min then
4:       min ← A[i]
5:    end if
6: end for
7: return min
```

n e l'altro con gli indici che variano da 1 a k. Quindi la complessità è al più $O(n*k)$ e quest'ultima coincide con $O(n)$, visto che k è costante. Inoltre la complessità è almeno $\Omega(n)$ in quanto abbiamo dei cicli for con indici che variano da 1 ad n che vengono sicuramente eseguiti.