Ordinamento

Sommario

- Ordinamento
  - Selection Sort
  - Bubble Sort/ Shaker Sort
  - Shell Sort

Cosa e' l'ordinamento

- Il problema consiste nell'elaborare insiemi di dati costituiti da record
- I record hanno sono costituiti da una *chiave* e (eventualmente) da altri *dati satellite*
- La chiave ha valori in un insieme *totalmente ordinato* (per cui vale la proprietà di tricomia cioè per ogni coppia di elementi a,b nell'insieme deve valere esattamente una delle seguenti relazioni: a=b, a<b, a>b)
- L'obiettivo dei metodi di ordinamento consiste nel riorganizzare i dati in modo che le loro chiavi siano disposte secondo un ordine specificato (generalmente numerico o alfabetico)

Perche' e' importante l'ordinamento

- L'ordinamento e' un passo intermedio utile per l'ottimizzazione di altr procedure molto comuni in vari algoritmi
  - Ricerca e fusione (merge)
  - Canonizzazione (trasformare un dato che puo' avere piu' di una possibile rappresentazione in una unica forma)
  - Comprensibilita' per lettori umani
- Una alta percentuale del tempo di esecuzione di una applicazione complessa e' speso in operazioni di ordinamento
- L'ordinamento e' in genere una sub-routine annidata profondamente all'interno di procedure iterative e dunque migliorarne l'efficienza ha profonde implicazioni sull'efficienza complessiva dei programmi

Tipi di ordinamento: interno/esterno

- Si distinguono metodi interni ed esterni:
  - *interni*: se l'insieme di dati è contenuto nella memoria principale
  - *esterni*: se l'insieme di dati è immagazzinato su disco o nastro
- Per metodi interni è possibile l'accesso casuale ai dati, mentre per i metodi esterni è possibile solo l'accesso sequenziale o a blocchi di grandi dimensioni
- Si usano i metodi esterni quando non vale l'ipotesi del calcolatore RAM con memoria illimitata

Tipo di ordinamento: stabili/non stabili

- Si distinguono i metodi di ordinamento in *stabili* o *non stabili*.
- Un metodo di ordinamento si dice stabile se preserva l'ordine relativo dei dati con chiavi uguali all'interno della sequenza da ordinare

Tipo di ordinamento *stabile*

- **Esempio:** se si usa un metodo stabile per ordinare per anno di corso una lista di studenti già ordinata alfabeticamente, otterremo una lista in cui gli studenti dello stesso anno sono ordinati alfabeticamente

```
2004 Alessandro
2005 Alessio
2003 Antonio
2004 Beppe
2003 Bruno
2003 Carlo
2004 Cristiano
2005 Dario
2005 Emanuele
```



```
2003 Antonio
2003 Bruno
2003 Carlo
2004 Alessandro
2004 Beppe
2004 Cristiano
2005 Alessio
2005 Dario
2005 Emanuele
```

Tipo di ordinamento: diretto/indiretto

- Si distinguono i metodi di ordinamento in *diretti* o *indiretti*.
- Un metodo di ordinamento si dice diretto se accede all'intero record del dato da confrontare, indiretto se utilizza dei riferimenti (puntatori) per accedervi

- Metodi indiretti sono utili quando si devono ordinare dati di grandi dimensioni
- In questo modo non è necessario spostare i dati in memoria ma solo i puntatori ad essi.

Tipo di ordinamento: sul posto/non sul posto

- Si distinguono i metodi di ordinamento *sul posto (in-place)* e non, che fanno cioè uso di *strutture ausiliare*
- Un metodo si dice che ordina sul posto se durante l'elaborazione riorganizza gli elementi del vettore in ingresso all'interno del vettore stesso
- Se il metodo, per poter operare, ha necessità di allocare un vettore di appoggio dove copiare i risultati parziali o finali dell'elaborazione (della stessa dimensione del vettore in ingresso) abbiamo il secondo caso

Selection Sort

- E’ uno degli algoritmi più semplici
- Il principio è:
  - si determina l’elemento più piccolo di tutto il vettore
  - lo si scambia con l’elemento in prima posizione del vettore
  - si cerca il secondo elemento più grande
  - lo si scambia con l’elemento in seconda posizione del vettore
  - si procede fino a quando l’intero vettore è ordinato
- Il nome deriva dal fatto che si seleziona di volta in volta il più piccolo elemento fra quelli rimanenti

Pseudocodice per SelectionSort

```
SelectionSort(A)
1  for i ← 1 to length[A]
2    do min ← i
3       for j ← i+1 to length[A]
4       do if A[j] < A[min]
5          then min ← j
6       A[i] ↔ A[min]
```

Caratteristiche del SelectionSort

- Il tempo di calcolo è $T(n)= \Theta(n^2)$
  - per ogni dato di posizione *i* si eseguono *n-1-i* confronti
  - il numero totale di confronti è pertanto (posto *j= n-1-i*)

$$\sum_{j=n-1..1} j = \sum_{j=1..n-1} j = n(n-1)/2 = \Theta(n^2)$$

- Più precisamente il Selection Sort effettua
  - circa $n^2/2$ confronti
  - *n* scambi

Caratteristiche del SelectionSort

- Uno svantaggio è che il tempo di esecuzione non dipende (in modo significativo) dal grado di ordinamento dei dati iniziali
- Un vantaggio è che ogni elemento è *spostato* una sola volta.
  - Se è necessario spostare i dati, allora per dati molto grandi questo è l'algoritmo che asintoticamente effettua il minor numero di spostamenti possibili.
  - Se il tempo di spostamento è dominante rispetto al tempo di confronto diventa un algoritmo interessante

BubbleSort

- E’ un metodo elementare
- Il principio di funzionamento è:
  - si attraversa il vettore scambiando coppie di elementi adiacenti
  - ci si ferma quando non è più richiesto alcuno scambio
- Il nome deriva dal seguente fenomeno:
  - quando durante l’attraversamento si incontra l’elemento più piccolo non ancora ordinato questo viene sempre scambiato con tutti, affiorando fino alla posizione giusta come una bolla
  - nel processo gli elementi maggiori *affondano* e quelli più leggeri *salgono a galla*

PseudoCodice per il BubbleSort

```
BubbleSort(A)
1  for i ← 1 to length[A]
2  do for j ← length[A] downto i+1
4     do if A[j-1] > A[j]
5          then A[j-1] ↔ A[j]
```

Caratteristiche del BubbleSort

- Il tempo di calcolo è $T(n)= \Theta(n^2)$
  - per ogni dato di posizione *i* si eseguono *n-1-i* confronti e *n-1-i* scambi
  - il numero totale di confronti è pertanto (posto *j= n-1-i*)

$$\sum_{j=1..n-1} j = n(n-1)/2 = \Theta(n^2)$$

- Il Bubble Sort effettua
  - circa $n^2/2$ confronti
  - $n^2/2$ scambi
- In generale è peggiore del selection sort
- **Nota:** si può migliorare interrompendo il ciclo più esterno qualora non si siano verificati scambi

Shaker Sort

- Come il Bubble Sort ma alternando passate da sinistra a destra e da destra a sinistra.
- In questo modo sia gli elementi pesanti affondano che affiorano quelli leggeri

Insertion Sort

- Lo abbiamo gia' visto
- Ha complessita' quadratica
- Tuttavia il numero di confronti e scambi dipende dal grado di ordinamento dei dati: il caso ottimo ha complessita' lineare
- Diventa interessante quando i dati sono parzialmente ordinati

Insertion Sort

```
INSERTION-SORT(A)
1  for j ← 2 to lenght[A]
2       do     key ← A[j]
3              i ← j - 1
4              while i>0 e A[i]>key
5                     do     A[i+1] ← A[i]
6                            i ← i - 1
7              A[i+1] ← key
```

Shell Sort

- La lentezza dell'ordinamento per inserzione e' dovuta al fatto che le operazioni di scambio avvengono tra elementi adiacenti
- Se l'elemento piu' piccolo e' alla fine dell'array ci vogliono N passi per disporlo al posto giusto
- L'idea dello shell sort e' di scambiare gli elementi prima molto distanti tra loro e poi progressivamente quelli piu' vicini

Shell Sort

- Per migliorare le cose si puo' lavorare considerando i dati in blocchi e ordinare per colonne
- Si inizia con molte colonne (elementi distanti) e si procede fino ad ottenere una unica colonna

```
3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2

3 7 9 0 5 1 6          3 3 2 0 5 1 5
8 4 2 0 6 1 5    ->    7 4 4 0 6 1 6
7 3 4 9 8 2            8 7 9 9 8 2

3 3 2          0 0 1
0 5 1          1 2 2
5 7 4          3 3 4
4 0 6    ->    4 5 6
1 6 8          5 6 8
7 9 9          7 7 9
8 2            8 9
```

Shell Sort

- In realta' non si dividono i dati in blocchi: si considerano nello stesso insieme i dati che hanno indici a distanza fissa.

- L'idea e' di ordinare l'array in modo che gli elementi che hanno una distanza h fra loro costituiscano una sequenza ordinata
- Un array che soddisfa questa proprieta' si dice h-ordinato o ordinato con passo h
- Se si eseguono piu' passate h-ordinando l'array prima con passo h grande e poi decrementandolo fino a passo 1 si ottiene un array via via sempre piu' ordinato fino ad averlo del tutto ordinato
- Visto che l'efficienza dell'insertion sort dipende da quanto e' gia' ordinato l'array otteniamo via via delle prestazioni migliori

Shell Sort

- L'idea e' di usare l'insertion sort per h-ordinare
- Basta sostituire gli incrementi/decrementi unitari con incrementi/decrementi di h posizioni
- La sequenza decrescente di valori di h viene calcolata a partire da un h grande (ex: N/9) con andamento esponenziale: h=h/3
- **Esempio:**
  - N=1000
  - h=111,37,12,4,1

Shell Sort (approssimato)

```
SHELL-SORT(A)
1  for h ← lenght[A]/9 to 0 with h ← h / 3
2  do  InsertionSort con passo h
```

Shell Sort

```
SHELL-SORT(A)
1  for h ← lenght[A]/9 to 0 with h ← h / 3
2  do   for j ← 1+h to lenght[A]
3       do     key ← A[j]
4              i ← j
5              while i > 1+h e A[i-h]>key
6                     do     A[i] ← A[i-h]
7                            i ← i - h
8              A[i] ← key
```

Complessita' dello Shell Sort

- Il caso peggiore rimane come per l'algoritmo da cui deriva (l'insertion sort) un $O(n^2)$
- Il caso medio e' difficile da calcolare perche' dipende dalla sequenza degli h-ordinamenti
- Questo e' un esempio di algoritmo semplice con proprieta' complesse
- Con sequenza di tipo:
  - Pratt: 1, 2, 3, 4, 6, 8, 9, 12, 16, ...2^p3^q si ha $O(n (\log n)^2)$
  - Knuth: 1, 4, 13, 40, 121,... $(3^s-1)/2$ si ha $O(n^{3/2})$
  - Sedgewick: 1, 5, 19, 41, 109, 209, ... (non mostrata) si ha $O(n^{4/3})$ caso pessimo e $O(n^{7/6})$ caso medio!