Ordinamento ottimo

Sommario

- Ordinamento Ottimo O(n lg n)
- Ordinamento O(n)

Ordinamento Ottimo

- In un ordinamento per confronto si usa il confronto tra coppie di elementi per ottenere informazioni sull'ordine della sequenza degli elementi
- Dati due elementi *a* e *b* per determinare l'ordine relativo fra questi si deve eseguire uno dei seguenti confronti:
  - $a > b$
  - $a < b$
  - $a \leq b$
  - $a \geq b$
- I confronti sono tutti equivalenti, nel senso che forniscono tutti la stessa informazione sull'ordinamento relativo tra a e b
- Consideriamo pertanto solo $a \leq b$

Albero di decisione

- Un algoritmo di ordinamento per confronto può essere rappresentato a livello astratto come un *albero di decisione*
- Un albero di decisione rappresenta, con una struttura ad albero, i confronti eseguiti da un algoritmo su di una sequenza di ingresso di data dimensione
- Sulle foglie dell'albero sono indicate le permutazioni dell'ingresso che corrispondono ad un particolare ordinamento dei dati in ingresso cioè al risultato dell' algoritmo di ordinamento

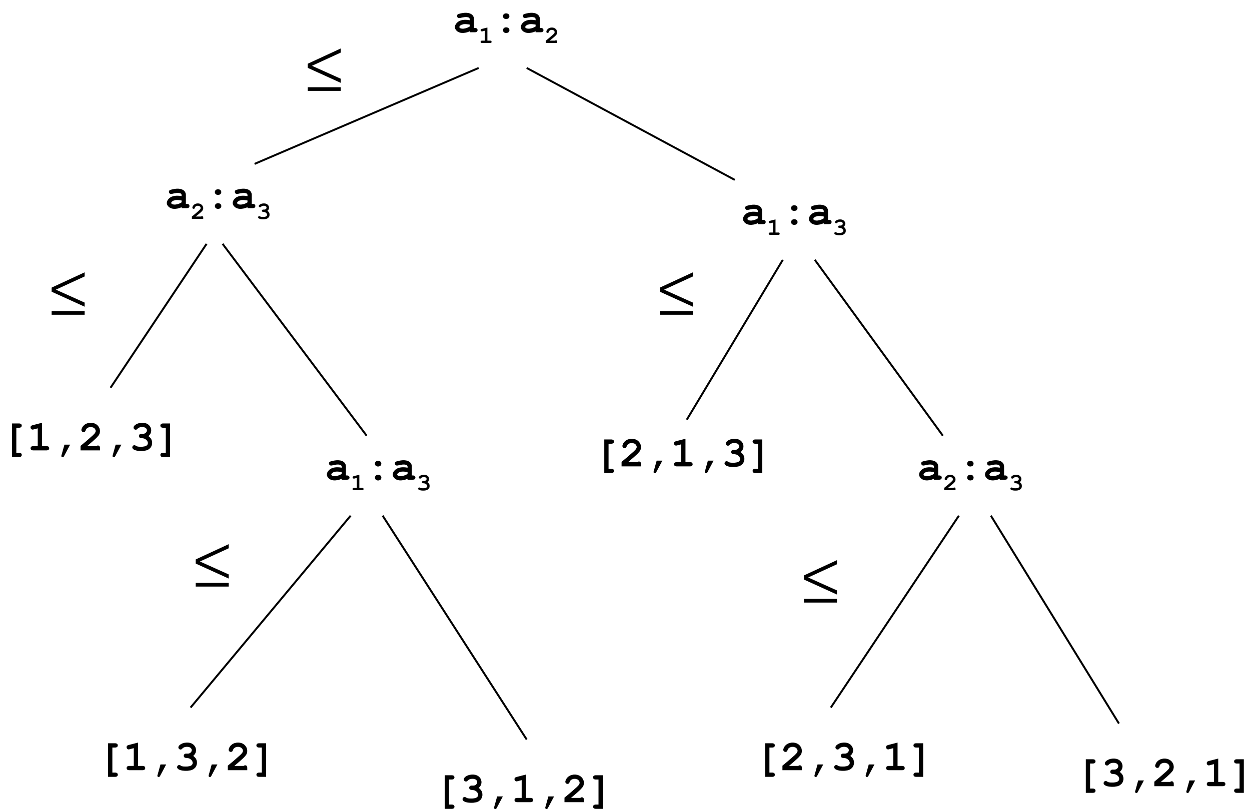
Albero di decisione

- In un albero di decisione si rappresentano solo i confronti eseguiti (niente istruzioni per il controllo, copia dati, etc)
- In ogni *nodo interno* si effettua uno ed un solo confronto fra *due* elementi
- Gli elementi confrontati sono indicati nella etichetta del nodo come a_i:a_j con i,j che variano nell'intervallo dell'indice degli elementi in ingresso
- Ogni *foglia* ha una etichetta del tipo [3,5,1,...] con la quale indichiamo una permutazione degli elementi in ingresso
- Con una permutazione indichiamo la sequenza degli indici degli elementi nel vettore originario seguendo la quale si ha la sequenza ordinata
- Ad es. nel caso [3,5,1,...] consideriamo come risultato prima il terzo elemento, poi il quinto, poi il primo, etc

Albero di decisione

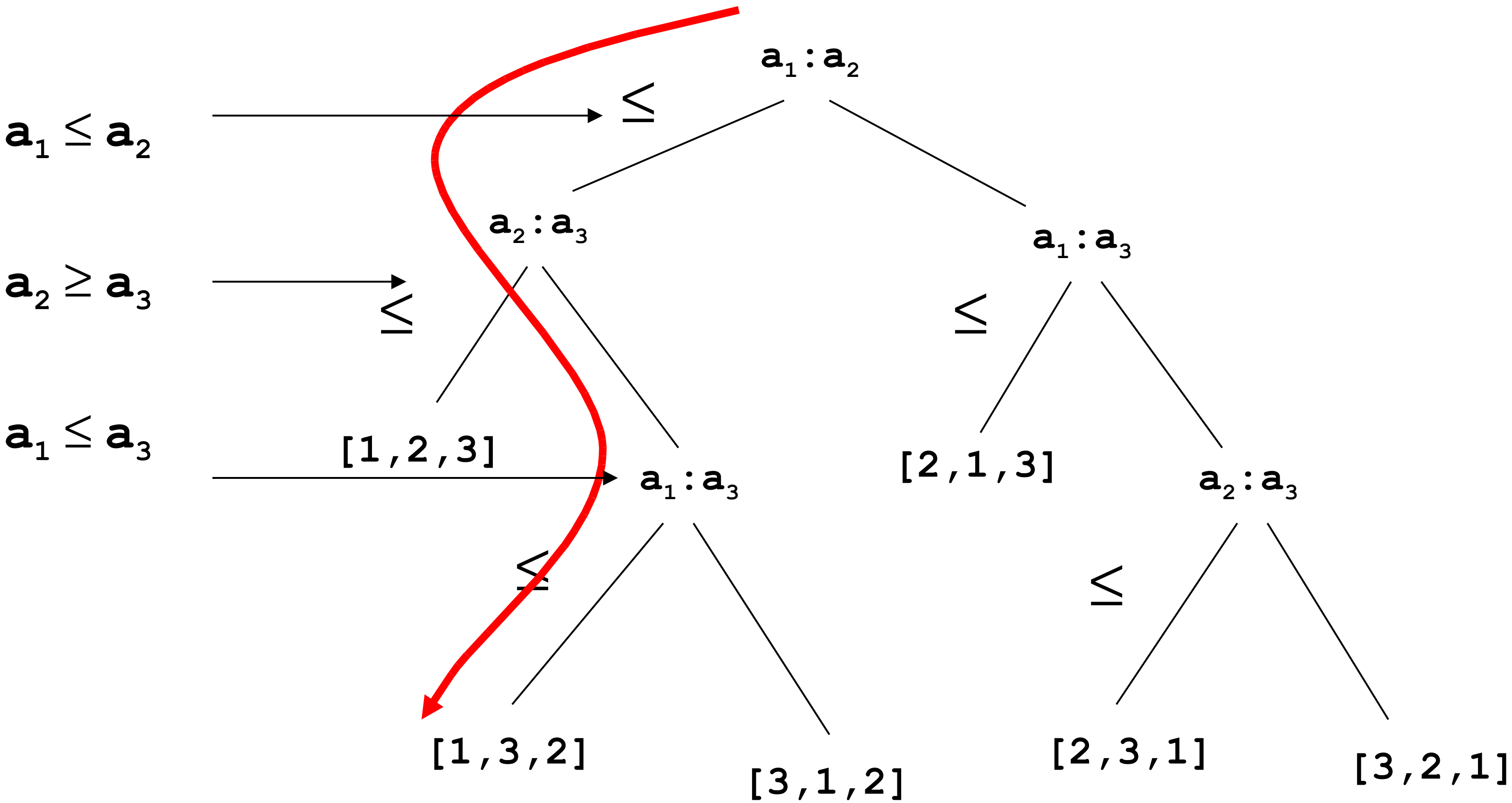
- Ad ogni nodo interno si compie un confronto e (per convenzione) se $a_i \leq a_j$ allora si scende nel figlio sinistro, altrimenti destro
- L'esecuzione di un algoritmo consiste nel compiere un cammino nell'albero di decisione a partire dalla radice fino ad una foglia
- Perché si abbia sempre una soluzione si deve garantire che ognuna delle n! permutazioni possibili dell'ingresso sia rappresentato come una foglia dell'albero di decisione

Albero di decisione per Ordinamento



Nota: 3 elementi, 3! combinazioni, 6 foglie

Esempio



Se $a_1 \le a_2$ e $a_1 \le a_3$ allora a_1 è l'elemento minore
e se $a_2 \ge a_3$ allora a_2 è l'elemento maggiore

Limite inferiore

- Il cammino più lungo in un albero di decisione dalla radice ad una qualunque foglia rappresenta il numero di confronti che l'algoritmo deve eseguire nel caso peggiore
- Il cammino più lungo è pari all'altezza dell'albero
- Un limite inferiore sull'altezza dell'albero di decisione di un algoritmo è dunque un limite inferiore sul tempo di esecuzione di un algoritmo di ordinamento per confronti
- Cioe' se determiniamo l'altezza minima di un albero di decisione per n elementi abbiamo il limite inferiore per il caso peggiore di un algoritmo di ordinamento per confronti su n elementi

Teorema sull'ordinamento ottimo

- **Teorema:**
  - Qualunque albero di decisione che ordina n elementi ha altezza $\Omega(n \ln n)$
- Dimostrazione:
  - Dato che vi sono n! permutazioni di n dati ed ogni permutazione rappresenta un possibile ordinamento allora l'albero binario di decisione deve avere n! foglie
  - Un albero binario di altezza h ha al più 2^h foglie (caso di albero binario completo)
  - Pertanto deve essere: $2^h \geq n!$
  - Usando l'approssimazione di Stirling: $n! \approx n^n$
  - Passando ai logaritmi: $\ln 2^h \geq \ln n^n$
  - ovvero: $h \geq n \ln n = \Omega(n \ln n)$

Conseguenze

- Un qualunque algoritmo di ordinamento per confronto non può avere una complessità asintotica inferiore a n ln n, ovvero si ha $T(n)=\Omega(n \ln n)$ per il ***caso peggiore***
- Ne consegue che il merge sort e lo heap sort sono algoritmi ottimi in quanto per il caso peggiore i limiti superiori $T(n)=O(n \ln n)$ corrispondono a quelli inferiori $\Omega(n \ln n)$

Ordinamento O(n)

- E’ tuttavia possibile scrivere algoritmi di ordinamento che operano in tempo O(n)
- Per farlo si deve abbandonare il metodo di ordinamento per confronto
- Se le chiavi da ordinare sono interi in un intervallo prefissato allora si può utilizzare direttamente il valore della chiave per posizionare l’elemento nella giusta posizione nel vettore ordinato finale

Counting Sort

- Il Counting Sort si basa sull'ipotesi che ognuno degli n elementi in ingresso sia:

  un intero nell'intervallo da 1 a k
- Se k=O(n) allora il tempo di esecuzione del CountingSort è O(n).
- Ovvero, se il valore massimo dei dati da elaborare è dello stesso ordine di grandezza della numerosità dei dati, allora il tempo di esecuzione del CountingSort è O(n).

Caratteristiche del Counting Sort

- L'algoritmo prende in ingresso un vettore, restituisce un secondo vettore ordinato ed utilizza un vettore di appoggio per l'elaborazione (ordinamento non in-place)
- L'algoritmo esegue un ordinamento stabile

Spiegazione Intuitiva di Counting Sort

- Per ogni elemento *x* dell'insieme da ordinare si determinano quanti elementi sono minori di *x*
- si usa questa informazione per assegnare ad *x* la sua posizione finale nel vettore ordinato
- se, ad esempio, vi sono 8 elementi minori di *x*, allora *x* andrà messo nella posizione 9
- bisogna fare attenzione al caso in cui vi siano elementi coincidenti. In questo caso infatti non vogliamo assegnare a tutti la stessa posizione.

Counting Sort

```
CountingSort(A,B,k)
for i← 1 to k
do    C[i] ← 0
for j← 1 to length[A]
do    C[A[j]] ← C[A[j]]+1
for i← 2 to k
do    C[i] ← C[i]+C[i-1]
for j ← length[A] downto 1
do    B[C[A[j]]] ← A[j]
      C[A[j]] ← C[A[j]]-1
```

Visualizzazione

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	0	2	3	0	1

	1	2	3	4	5	6
C	2	2	4	7	7	8

	1	2	3	4	5	6	7	8
B							4	

	1	2	3	4	5	6	7	8
B		1					4	

	1	2	3	4	5	6	7	8
B		1				4	4	

	1	2	3	4	5	6
C	2	2	4	6	7	8

	1	2	3	4	5	6
C	1	2	4	6	7	8

	1	2	3	4	5	6
C	1	2	4	5	7	8

	1	2	3	4	5	6	7	8
B	1	1	3	3	4	4	4	6

Tempo di calcolo del Counting Sort

- Esaminando l’algoritmo si osserva che vi sono due cicli di lunghezza k e due di lunghezza n
- Si può far vedere che la complessità è $\Theta(k+n)$
- se $k = \Theta(n)$ allora la complessità del Counting Sort è complessivamente $\Theta(n)$

Stabilità del Counting Sort

- L'algoritmo Counting Sort è un metodo di ordinamento stabile
- infatti elementi con lo stesso valore compaiono nel vettore risultato B nello stesso ordine che avevano nel vettore di ingresso A
- Nota: se invece di procedere dall'elemento di indice maggiore a quello minore durante l'assegnazione al vettore B, si procedesse dal minore al maggiore, si perderebbe la proprietà di stabilità

Radix Sort

- Il Radix Sort è un algoritmo di ordinamento usato per ordinare record con chiavi multiple
- Un esempio di record con chiavi multiple è dato dalla data gg/mm/aaaa. Per ordinare per data si deve ordinare l'anno e a parità di anno si deve ordinare per mese e a parità di mese per giorno
- Un altro esempio di record a chiave multipla è dato dal considerare le cifre di un intero come chiavi separate. Per ordinare interi si ordina per la cifra di posizione maggiore e in caso di parità per quelle di ordine via via minore

Spiegazione intuitiva

- Il Radix Sort opera in modo contro intuitivo ordinando prima sulle cifre meno significative e poi su quelle via via più significative
  - una persona ordinerebbe considerandi inizialmente le cifre piu' significative, ex tutti i numeri 1xx prima dei numeri 2xx

- Supponiamo di dover ordinare una sequenza di numeri a 3 cifre
- Utilizzando un ordinamento di tipo stabile possiamo procedere ordinando prima per le unità, poi le decine e in ultimo le centinaia
- ad ogni passo la stabilità ci garantisce che le cifre precedenti sono già ordinate

Esempio

Il dato 720 precede il dato 329:
a parità di chiave (2)
si mantiene l'ordine relativo

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839
Sequenza in ingresso	Ordinamento sull'ultima cifra	Ordinamento (stabile) sulla seconda cifra	

PseudoCodice

Nota: si suppone che i numeri siano rappresentati in un sistema posizionale, ovvero cifre di indice minore hanno peso minore.

```
Radix-Sort(A,d)
for i 1 to d
do   metodo di ordinamento stabile su cifra i
```

Tempo computazionale

- Il tempo di esecuzione dipende dall'algoritmo di ordinamento stabile scelto per ordinare le singole cifre
- se si usa il Counting Sort si ha che per ognuna delle d cifre si impiega un tempo $\Theta(k+n)$ pertanto si ha

$$\Theta(dk+dn)$$

- se d è una costante rispetto a n
- se $k=\Theta(n)$
- allora per il radix sort si ha $\Theta(n)$

Nota sulle prestazioni

- Se vogliamo ordinare 10^6 numeri a 3 cifre
  - con il radix sort si effettua per ogni dato 3 chiamate al counting sort, per un totale proporzionale a 3 n operazioni
  - con algoritmi O(n lg n) si ha equivalentemente (lg n=14) un costo proporzionale a 14 n operazioni
- Andando a estrarre le costanti numeriche nascoste nella notazione asintotica si vede che il radix sort può essere conveniente
- Lo svantaggio sta nel fatto che il metodo non è un ordinamento in loco e ha bisogno di più del doppio della memoria dei metodi in loco