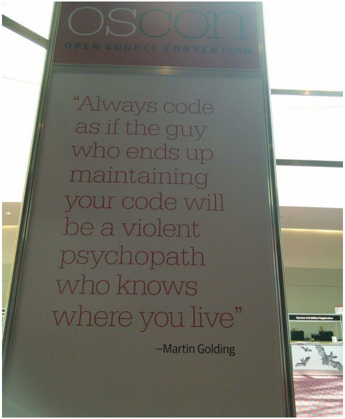
Algoritmi e strutture dati

CountingSort e *RadixSort*



Menú di questa lezione

In questa lezione vedremo, prima, i limiti inferiori per l'ordinamento basato su confronti, e poi gli algoritmi *CountingSort* e *RadixSort*, che utilizzano ipotesi aggiunttive per vincere questi limiti.

Limiti per l'ordinamento basato su confronti

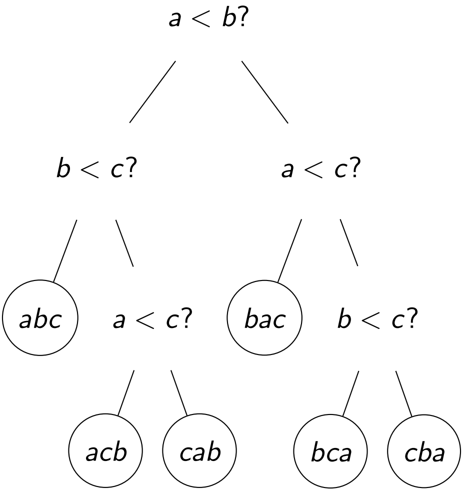
All'inizio del corso abbiamo definito i principi per il calcolo delle complessitá degli algoritmi, che naturalmente si applicano al calcolo delle complessitá dei problemi. In particolare, abbiamo introdotto il concetto di operazione elementare, e detto che calcolare la complessitá è semplicemente contare il numero di operazioni elementari. In questa parte affrontiamo la seguente domanda: qual è la complessitá **minima** del problema dell'ordinamento? Per rispondere, possiamo focalizzarci su un tipo specifico di operazione elementare; se riusciamo dire **questo è il minimo numero di operazioni di quel tipo**, allora abbiamo un limite inferiore per tutte le operazioni.

Limiti per l'ordinamento basato su confronti

Diciamo che un algoritmo di ordinamento si **basa sui confronti** se ogni passo (essenziale) puó essere visto come una operazione di confronto tra due elementi, seguita da uno spostamento. Osserviamo che tutti gli algoritmi di ordinamento visti fino ad ora, tanto elementari come non elementari, sono basati sul confronto. Più avanti ne vedremo incidentalmente ancora uno (*HeapSort*), che sarà ancora basato sui confronti. Possiamo **generalizzare** il processo di ordinare per confronti? Assumendo che possiamo solo confrontare 2 elementi per volta, il processo di ordinare puó essere visto come un albero binario (non confondiamoci con la nozione di albero come struttura dati, che vedremo più avanti) la cui radice è l'input. Ad ogni nodo si associa la **permutazione** dell'oggetto che corrisponde ad uno scambio.

Limiti per l'ordinamento basato su confronti

Vediamo un esempio con tre elementi a, b, c, che assumiamo tutti diversi. Basandoci sul confronto(a due a due) dobbiamo trovare il loro ordinamento corretto. Ad ogni passo, effettuiamo, se necessario, uno scambio.



Quante permutazioni possibili esistono per n elementi? Precisamente $n!$, e un albero binario con $n!$ foglie è alto, almeno, $log(n!)$. Quindi, $log(n!)$ è un limite inferiore alla lunghezza massima di un ramo nell'albero che rappresenta l'esecuzione di un qualsiasi algoritmo di ordinamento basato sui confronti. Sappiamo che

→ ESERCIZIO 6 12

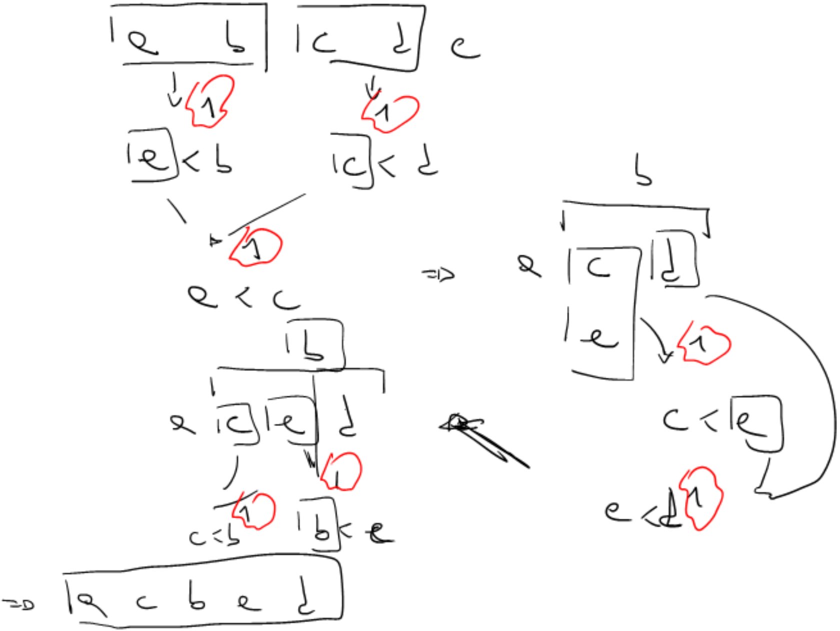
$$log(n!) = \Theta(n \cdot log(n))$$

Quindi, quindi il nostro limite inferiore è $\Omega(n \cdot log(n))$. Per esempio, questo significa che *MergeSort* è **ottimo**, visto che ha complessità, nel caso peggiore, $\Theta(n \cdot log(n))$. Osserviamo che questa è la prima volta che riusciamo dire qualcosa sulla complessità di un problema, e non solo su quella di un algoritmo. *QuickSort* non è ottimo, neppure nella sua versione randomizzata, ma altre considerazioni che abbiamo fatto ci portanto a preferirlo in tante situazioni. Attenzione: come sempre stiamo usando una notazione asintotica. Per numeri sufficientemente **piccoli**, possiamo fare meglio di $n \cdot log(n)$. In realtà, il limite che **non possiamo** vincere è $log(n!)$. La differenza non è piccola, come mostrato dal seguente esempio.

Limiti per l'ordinamento basato su confronti

É possibile ordinare correttamente 5 numeri basandoci sui confronti in meno di 8 confronti? Se rispondiamo no, allora sbagliamo, anche se è vero che $5 \cdot log(5) = 11{,}6 \approx 12$. Questo ragionamento non terrebbe conto del fatto che il limite inferiore dato è asintotico. La risposta è sì (almeno in linea di principio), perchè $log(5!) = log(120) = 6{,}90 \approx 7$, che è il numero minimo di confronti necessari per decidere l'ordine di 5 elementi diversi. La letteratura ci presenta algoritmi specifici per ordinare un numero piccolo ed esatto di elementi, ad esempio 5. In linea di principio ne esiste uno specifico per qualunque numero **fissato** di elementi: questi algoritmi non sono eleganti, non insegnano nulla di concettuale, e servono esclusivamente a rappresentare una idea.

Ordiniamo 5 elementi a, b, c, d, e



Limiti per l'ordinamento basato su confronti

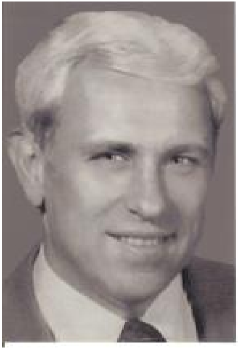
Consideriamo infatti tre elementi a, b, c, tali che $a < b < c$, ed un quarto elemento d che vogliamo inserire nella terna ordinata. Quanti confronti sono necessari? La risposta è due: prima confronto d con b, poi con a oppure con c, a seconda del risultato precedente. Questa osservazione ci fornisce l'idea per un algoritmo **specifico** per ordinare 5 elementi con 7 confronti: dati a, b, c, d, e, prima confrontiamo a con b, e parallelamente c con d, e supponiamo $a < b$ e $c < d$ (2 confronti); poi confrontiamo a con c, e supponiamo $a < c$ (1 confronto); poi ordinamo e nella sequenza acd, e supponiamo $c < e < d$ (2 confronti); finalmente, ordinamo b nella sequenza ced (2 confronti).

Limiti per l'ordinamento basato su confronti

Riassumendo, siamo stati capaci, per questo particolare problema, di dire che esiste un numero minimo di confronti che qualunque algoritmo deve fare per poter risolvere il problema dell'ordinamento (basandosi sui confronti). Pertanto, esiste un numero minimo di operazioni, e quindi una complessitá minima. Questo è un risultato importante, perchè, solitamente, non abbiamo limiti minimi di complessitá a questo livello di dettaglio (tranne quelli triviali). Come abbiamo detto, contare operazioni concrete, invece di operazioni qualsiasi, è una strategia che possiamo anche applicare in altri contesti quando calcoliamo complessitá esatte.

Ipotesi aggiuntive

Aggiungendo delle ipotesi agli oggetti che si vogliono ordinare, peró, possiamo vincere i limiti. Un esempio che vediamo è quello degli algoritmi *CountingSort* e *RadixSort*, proposti da Harold Seward attorno al 1950.



Ipotesi aggiuntive

I due algoritmi sono legati tra loro, sebbene possano essere visti separatamente. Entrambi sono basati sull'ipotesi aggiuntiva di stare ordinando numeri interi di cui conosciamo il limite superiore. *CountingSort* lavora direttamente con numeri, come gli altri algoritmi che abbiamo visto, e *RadixSort* ne generalizza l'utilizzo. Assumiamo dunque che ogni elemento dell'input sia un intero che varia tra 0 e k, dove k è, a sua volta, un intero. Questa ipotesi non è cosí rara nella realtá, quando usiamo un algoritmo di ordinamento per scopi specifici. *CountingSort*, in particolare, si basa su un array A, su uno di appoggio C (quindi, non è in place), e su uno di uscita B, dove si ottiene il risultato.

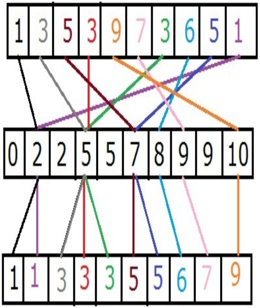
Ordinamento con *CountingSort*

→ Alloca Array di contatori

→ Azzera tutti i contatori

proc ***CountingSort*** (A, B, k)

- **let** $C[0, \ldots, k]$ **new array**
- **for** $(i = 0$ **to** $k)$ $C[i] = 0$
- **for** $(j = 1$ **to** $A.length)$ $C[A[j]] = C[A[j]] + 1$
- **for** $(i = 1$ **to** $k)$ $C[i] = C[i] + C[i-1]$
- **for** $(j = A.length$ **downto** $1)$
  - $B[C[A[j]]] = A[j]$
  - $C[A[j]] = C[A[j]] - 1$



→ Conta tutti gli elementi uguali ad ogni A[i] nel contatore C[A[i]]

→ Conta tutti gli elementi minori o uguali ad ogni A[i] in C[A[i]]

→ Riposiziona al posto giusto in B

Counting sort usa gli elementi di A come indici

A: | 7 | 1 | 4 | 5 | 6 | 6 | 9 |

1 2 3 4 5 6 7

→ Nell'array ordinato hai delle molte posizioni tante che tutti i più piccoli di 4 vanno a sx

=> Basta contare gli elementi più piccoli di un certo elemento per sapere la sua posizione

C: | 0 | 1 | 0 | 0 | 1 | 1 | 2 | 1 | 0 | 0 | 1 |

0 1 2 3 4 5 6 7 8 9

C': | 0 | ~~1~~ | 1 | 1 | ~~2~~ | ~~3~~ | ~~5~~ | 6 | 6 | ~~7~~ 6 |

0 1 2 3 4 5 6 7 8 9

=> Confronti

B: | 1 | 4 | 5 | 6 | 6 | 7 | 9 |

1 2 3 4 5 6 7

Correttezza di *CountingSort*

Come sempre, vediamo la correttezza dell'algoritmo. Per la **terminazione**, vediamo che tutto l'algoritmo è governato da cicli **for**, e quindi termina per definizione. Guardiamo la **correttezza**. I primi tre cicli sono abbastanza immediati da vedere: alla fine del secondo ciclo **for**, in C la posizione i-esima contiene il numero di elementi di A che sono uguali ad i, e alla fine del terzo, la posizione i-esima contiene il numero di elementi di A che sono uguali ad i. Un'**invariante** corretta per il quarto ciclo **for** è: al j-esimo passo (con j che va da n a 1), $C[A[j]]$ è la posizione corretta di $A[j]$ in B. In primo luogo osserviamo che, prima di iniziare il quarto ciclo, per ogni j si ha che $C[A[j]] - z$ è la posizione corretta di $A[j]$ in B se $A[j]$ è tale che $|\{A[l] \mid l > j, A[l] = A[j]\}| = z$.

Correttezza di *CountingSort*

Adesso dimostriamo il caso generale.

- **Caso base**. L'invariante è chiaramente vera per $j = n$, per l'osservazione precedente.
- **Caso induttivo**. Supponiamo che l'invariante sia vera per un certo j. Dopo aver effettuato l'inserimento di $A[j]$ nella posizione $C[A[j]]$, quest'ultimo valore diminuisce di una unitá. Consideriamo adesso l'elemento $A[j-1]$. Se questo è diverso da tutti gli elementi $A[l]$ con $l \geq j$, allora per l'osservazione precedente l'invariante è ancora vera. Supponiamo invece che esistano p elementi del tipo $A[l]$, con $l \geq j$, tali che $A[l] = A[j-1]$. Nei passaggi precedenti, il valore $C[A[j-1]]$ è dunque diminuito di p unità. Questo significa che adesso il valore di $C[A[j-1]]$ è quello della posizione corretta di $A[j-1]$ in B, come volevamo.

La correttezza di questa invariante implica direttamente che l'ultimo ciclo dell'algoritmo ordina A in B, come richiesto.

Complessità di *CountingSort*

La **complessità** di *CountingSort* è data dai quattro cicli. Due di questi hanno complessitá $\Theta(n)$ e gli altri due hanno complessitá $\Theta(k)$. Pertanto, se $k = O(n)$ la complessitá totale è $\Theta(n)$. Ma essendo precisi, la complessità andrebbe scritta come $\Theta(n + k)$ (nella notazione asintotica, il '$+$' si legge come 'il massimo tra'). Quindi se volessimo usare *CountingSort* su un array qualsiasi che contiene interi (o chiavi tradotte ad interi), di cui non conosciamo il massimo, l'idea di fare una passata iniziale per computare il massimo, e dichiarare C in maniera dinamica, è rischiosa: se k è troppo grande, l'allocazione potrebbe fallire, e comunque, se $k >> n$ (leggi: molto maggiore di), la complessitá non è piú lineare.

L'algoritmo *RadixSort* per ordinare elementi multi-indice

Immaginiamo adesso di voler ordinare n elementi multi-indice. Un esempio potrebbe essere quello di ordinare n date (giorno-mese-anno). Chiamiamo d il numero di indici, e assumiamo che sia fisso. Chiaramente, potremmo convertire ogni elemento in un unico intero e usare uno dei metodi giá visti. Ma sotto le stesse ipotesi di *CountingSort*, possiamo ottenere migliori risultati? Per avere un'idea chiara, pensiamo ad un esempio con $d = 3$, tale che ogni indice varia tra 0 e 9, e quindi abbiamo n elementi ognuno dei quali è un oggetto da 0 a 9-9-9.

ESEMPIO DI ORDINAMENTO MULTI INDICE

7/3/1995	1/10/1974	5/1/1996	5/2/1957
5/1/1996	1/3/1974	5/2/1957	1/3/1974
1/10/1974	3/10/1995	1/3/1974	1/10/1974
1/3/1974	5/1/1996	7/3/1995	7/3/1995
5/2/1957	5/2/1957	1/10/1974	3/10/1995
3/10/1995	7/3/1995	3/10/1995	5/1/1996

- SIG   + SIG

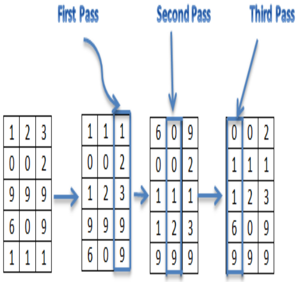
L'algoritmo *RadixSort* per ordinare elementi multi-indice

Un algoritmo classico che si puó utilizzare per risolvere questo problema prevede di ordinare gli elementi rispetto all'indice piú significativo, per poi passare al secondo, e cosí via. Nell'esempio, avremmo le date ordinate per anno, e quindi mese, ed infine giorno. Il difetto principale di questa strategia è che dopo la prima passata di ordinamento dobbiamo separare gli elementi per gruppi (identificati dagli indici piú significativi secondo i quali abbiamo giá ordinato) per effettuare la seguente. Quando disponiamo di un algoritmo stabile come *CountingSort* possiamo usare una strategia contro-intuitiva: ordinare prima secondo gli indici meno significativi.
RadixSort **non** è un algoritmo a sè stante, ma un **meta-algoritmo**: si basa su un algoritmo interno di ordinamento, per esempio *CountingSort*.

L'algoritmo *RadixSort* per ordinare elementi multi-indice

UN QUALUNQUE ALG.
D. ORDINAMENTO

proc ***RadixSort*** (A, d)
$\{$**for** $(i = 1$ **to** $d)$ $\;AnyStableSort(A)$ *on digit* i



Correttezza di *RadixSort*

Ragioniamo come sempre. La **terminazione** è completamente ovvia, assumendo che la procedura interna, qualunque essa sia, termini. Per la **correttezza**, l'**invariante** è: dopo la i-esima esecuzione del ciclo piú interno, gli elementi formati dalle ultime i colonne sono correttamente ordinati.

i colonne meno significative

Correttezza di *RadixSort*

Mostriamo che l'invariante funziona.

- **Caso base**. Se $i = 1$, allora stiamo parlando di elementi fatti da un solo indice. Quindi, la correttezza segue dalla correttezza dell'algoritmo stabile utilizzato come sotto-procedura.
- **Caso induttivo.** Assumiamo allora che $i > 1$, e che l'invariante valga per $i - 1$. Consideriamo l'indice i-esimo. Dopo la i-esima esecuzione del ciclo **for**, gli elementi sono lessicograficamente ordinati per sull'indice i-esimo (il piú significativo). Consideriamo due elementi a, b che hanno lo stesso valore per l'indice i-esimo, ma valori diversi sull'indice $(i - 1)$-esimo (senza perdita di generalitá, $a < b$ sull'indice $i - 1$). Per ipotesi induttiva, dopo la $(i - 1)$-esima esecuzione a viene posto prima di b; poichè la sotto-procedura è stabile, questa relazione viene mantenuta dopo la i-esima esecuzione, implicando che vale ancora $a < b$ dopo la i-esima esecuzione.

uso l'ipotesi di
stabilità

Complessitá di *RadixSort*

Tipicamente, ogni indice considerato da solo è numerico con massimo k costante (per esempio, nelle date, i giorni vanno da 1 a 31). Quindi, tipicamente, si usa *CountingSort* come procedura interna. Pertanto, la complessitá di *RadixSort* per n elementi su d indici, essendo ogni indice limitato tra 0 e k, è $\Theta(d \cdot (n + k))$. In generale, peró, la complessitá nel caso pessimo è $O(d \cdot f(n))$, dove $O(f(n))$ è la complessitá del caso pessimo della procedura interna.

Conclusione

Il trattamento degli algoritmi non basati sul confronto termina la nostra panoramica sull'ordinamento, anche se piú avanti vedremo ancora un algoritmo di ordinamento, ma in un altro contesto. Gli ordinamenti per array di lunghezza fissata sono estremamente complessi dal punto di vista del codice quando questi superano qualche unitá, ma sono estremamente utili per le ottimizzazioni di carattere euristico.