Appendice A

Algoritmi di ordinamento

Introduzione

Durante il corso abbiamo visto alcuni algoritmi di ordinamento, le loro complessità sono riassunte nella tabella A.1.

Tabella A.1: Complessità degli algoritmi di ordinamento

Algoritmo	Caso ottimo	Caso pessimo	Tecnica utilizzata
SelectionSort	$\Omega(n)$	$\mathcal{O}(n)$	-
InsertionSort	$\Omega(n)$	$\mathcal{O}(n^2)$	-
ShellSort	$\Omega(n)$	$\mathcal{O}(n^{3/2})$	ricerca locale
MergeSort	$\Theta(n \log n)$		dividi-et-impera
HeapSort	$\Theta(n \log n)$		-
QuickSort	$\Omega(n \log n)$	$\mathcal{O}(n^2)$	-

shellSort ha una complessità "migliore" di insertionSort $n^{3/2} = n\sqrt{n}$). Sappiamo inoltre che quickSort nel caso medio ha una complessità di $n \log n$.

Possiamo arrivare alla conclusione che tutti questi algoritmi sono *basati su confronti*: le decisioni sull'ordinamento vengono prese in base al confronto ($<$, $=$, $>$) fra due valori. Esistono altri modi per ordinare gli oggetti, ma gli algoritmi generali devono essere necessariamente basati sui confronti.

Gli algoritmi migliori che abbiamo visto finora hanno una complessità di $\mathcal{O}(n \log n)$, insertionSort e shellSort sono più veloci ($\Omega(n)$ solo in casi speciali, è quindi lecito chiederci se sia possibile fare meglio di così.

Dimostrazione del limite inferiore del problema dell'ordinamento

È possibile dimostrare che qualunque algoritmo di ordinamento *basato sui confronti* ha una complessità di $\Omega(n \log n)$.

Partiamo facendo alcune assunzioni. Consideriamo un qualunque algoritmo basato su confronti. Assumiamo che tutti i valori siano distinti (non abbiamo perdita di generalità). L'algoritmo può essere quindi rappresentato tramite un *albero di decisione*, ossia un albero binario che rappresenta i confronti con gli elementi.

Gli alberi di decisione hanno due proprietà:

1. il *cammino radice-foglia* rappresenta la sequenza di confronti eseguiti dall'algoritmo corrispondente;
2. l'*altezza dell'albero* rappresenta il numero di confronti eseguiti dall'algoritmo corrispondente nel caso pessimo.

Si considerino tutti gli alberi di decisione ottenibili da algoritmi di ordinamento basati su confronti.

Lemma 1. *Un albero di decisione per l'ordinamento di n elementi contiene **almeno** $n!$ foglie.*

Questo accade perché le foglie rappresentano il modo in cui dobbiamo scambiare i valori e siccome $n!$ è una qualsiasi delle permutazioni in cui riceviamo il nostro input, l'albero di decisione corrispondente deve avere una foglia per ogni suo elemento. Potrebbe averne anche più di una foglia, in quanto l'algoritmo potrebbe non essere efficiente ed effettuare più ordinamenti di quanto ne siano necessari, ma deve averne *almeno* $n!$.

Ora possiamo restringerci al caso dell'albero binario in quanto abbiamo valori distinti e lo possiamo fare senza perdere generalità.

Lemma 2. *Sia T un albero binario in cui ogni nodo interno ha esattamente 2 figli (< e >) e sia k il numero delle sue foglie. Allora l'altezza dell'albero è* ***almeno*** $\log k$ *ovvero* $\Omega(n \log n)$.

L'altezza sarà $\log k$ solo se l'albero è perfettamente bilanciato.

Teorema 3. *Il numero di confronti necessari per ordinare n elementi nel caso peggiore è* $\Omega(n \log n)$.

Abbiamo $n!$ foglie, quindi l'altezza dell'albero è $\sim \log n!$ e abbiamo che $\sim n \log n$.

A.1 Algortimi non basati sui confronti

A.1.1 Spaghetti Sort

L'algoritmo spaghettiSort è caratterizzato da 5 fasi:

1. prendi n spaghetti;
2. taglia lo spaghetto i-esimo in modo proporzionale all'i-esimo valore da ordinare;
3. con la mano, afferra gli n spaghetti e appoggiali verticalmente sul tavolo;
4. prendi il più lungo, misuralo e metti il valore corrispondente in fondo al vettore da ordinare;
5. ripeti il passo 4 fino a quando non hai terminato gli spaghetti.

Nel modo in cui è stato descritto questo algoritmo è infinitamente parallelo: qualsiasi sia il numero di spaghetti potremmo trovare lo spaghetto più lungo in $\mathcal{O}(1)$. Quindi questo algoritmo ha una complessità di $\mathcal{O}(n)$.

A.1.2 Counting Sort

Assumiamo che i numeri da ordinare siano compresi in un intervallo $[1 \dots k]$ (questo algoritmo non funziona con le stringhe).

Come funziona Costruisce un vettore di appoggio $B[1 \dots k]$ che conta il numero di volte che un valore compreso in $[1 \dots k]$ compare in A. Ricolloca i valori così ottenuti nel vettore da ordinare A.

Nota. L'intervallo non deve necessariamente iniziare in 1 e finire in k; qualunque intervallo di cui conosciamo gli estremi può essere utilizzato nel Counting Sort.

Algoritmo 1: Algoritmo di ordinamento Counting Sort

```
countingSort(ITEM[] A, int n, int k)
    int[] B ← new int[1...k] // creo il vettore d'appoggio
    for i ← 1 until k do // O(k)
        B[i] = 0 // azzero il vettore d'appoggio
    for j ← 1 until n do // O(n)
        B[A[j]] = B[A[j]] + 1 // conto quante volte incontro quel valore
    // vado a ricollocare i vari valori man mano che li incontro
    j = 1 // partendo dalla prima posizione
    from i = 1 until k do // scorro tutto il vettore
        while B[i] > 0 do // per tutte le occorrenze di un determinato valore
            A[j] ← i // lo inserisco nel vettore
            j++ // scorro il cursore
            B[i] ← B[i] − 1 // diminuisco le occorrenze
```

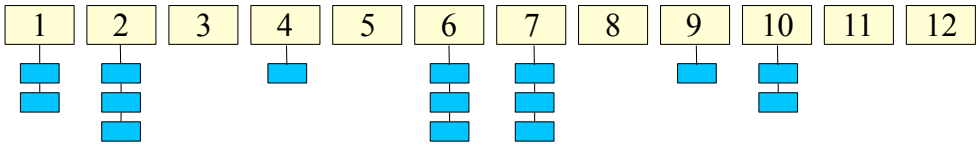
Complessità Inizializzare il vettore d'appoggio costa $\mathcal{O}(k)$. Contare le occorrenze di un valore costa $\mathcal{O}(n)$, in quanto fare riferimento ad una casella di memoria dato un valore costa $\mathcal{O}(1)$ e lo faccio n volte. L'algoritmo ha una complessità totale di $\mathcal{O}(n+k)$. Se k è $\mathcal{O}(n)$, allora la sua complessità è $\mathcal{O}(n)$.

Questo algoritmo non è basato sui confronti. Abbiamo quindi cambiato le condizioni di base e la dimostrazione che abbiamo visto per il limite inferiore non vale. Ad esempio se k è $\mathcal{O}(n^3)$ questo algoritmo è peggiore di tutti quelli visti finora.

A.1.3 Pigeonhole Sort

Se dobbiamo ordinare chiavi numeriche basse e valori associati, allora possiamo usare pigeonholeSort. Sfruttiamo lo stesso meccanismo di countingSort, ma anziché contare le occorrenze le inseriamo all'interno di una lista concatenata. L'inserimento di un elemento in una lista ha costo $\mathcal{O}(1)$, quindi l'algoritmo ha $\mathcal{O}(n)$ a patto che sia possibile limitare l'input a k elementi (possibilmente piccolo). Il costo complessivo risulta quindi $\mathcal{O}(n+k)$ e vale considerazione precedente.so

Nota. pigeonholeSort è un'estensione del countingSort che permette di ordinare in tempo lineare coppie (chiave, valore), invece che singoli interi. Le chiavi devono essere comprese fra 1 e k (con k possibilmente piccolo).



A.1.4 Bucket Sort

Se i valori in input sono:

1. valori reali uniformemente distribuiti nell'intervallo $[0, 1)$;
2. oppure è possibile normalizzarli nell'intervallo $[0, 1)$ in tempo lineare, in quanto sono uniformemente distribuiti, allora

è possibile usare bucketSort, ossia una versione di pigeonholeSort dove utilizziamo dei concetti probabilistici (l'ipotesi di uniformità) per sapere quanti elementi ci aspettiamo che siano contenuti in ogni lista.

Come funziona Divide l'intervallo in n sottovettori di dimensione $1/n$, detti *bucket*, e poi distribuisce gli n numeri nei *bucket*. I valori così inseriti possono essere riordinati tramite insertionSort.

La complessità attesa è di $\mathcal{O}(n)$, molto conveniente, ma l'ipotesi sui valori dell'input è molto stringente (ossia che siano uniformente distribuiti).

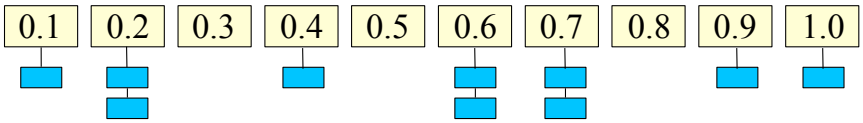


Figura A.1: Il primo *bucket* conterrà gli elementi con valore contenuto nell'intervallo $[0, 0.1)$, il secondo con valori contenuti nell'intervallo $[0.1, 0.2)$ e così via. Per l'ipotesi di uniformità, il numero atteso di valori nei bucket è 1.

Proprietà degli algoritmi di ordinamento

Definizione (stabilità)**.** Un algoritmo di ordinamento è detto **stabile** se preserva l'ordine iniziale tra due elementi con la stessa chiave.

Algoritmi stabili insertionSort, mergeSort, pigeonholeSort sono algoritmi stabili, mentre heapsort (non c'è modo di renderlo stabile) e quickSort (esistono delle versioni complicate che lo rendono tale) non lo sono. È possibile rendere stabile mergeSort avendo l'accortezza di inserire ordinatamente i valori nel vettore al momento dell'unione dei sottovettori.

Nota. Qualunque algoritmo di ordinamento può essere reso stabile. Basta associare al valore da ordinare l'indice in cui si trova all'inizio dell'ordinamento. Ordinando quindi prima per valore e poi per posizione.

A.2 Rissunto algoritmi di ordinamento

insertionSort ha una compleesità di $\Omega(n)$ nel caso ottimo e di $\mathcal{O}(n^2)$ nel caso pessimo. È stabile, sul posto, iterativo. Adatto per piccoli valori (non utilizzare quickSort) e sequenze quasi ordinate.

mergeSort ha una complessità di $\Theta(n \log n)$, è stabile ma richiede $\mathcal{O}(n)$ spazio aggiuntivo, è ricorsivo (richiede $\mathcal{O}(\log n)$ spazio nello *stack*). Buona performance in *cache*, buona parallelizzazione.

heapsort ha una complessità di $\Theta(n \log n)$, non stabile, sul posto, iterativo. Cattiva performance in cache, cattiva parallelizzazione viene quindi preferito in sistemi embedded dove parallelizzazione e memoria aggiuntiva non vengono considerati.

quickSort $\mathcal{O}(n \log n)$ in media, $\mathcal{O}(n^2)$ nel caso peggiore, non stabile, ricorsivo (richiede $\mathcal{O}(\log n)$ spazio nello *stack*). Buona performance in cache, buona parallelizzazione, buoni fattori moltiplicativi. È una delle scelte migliori grazie al meccanismo di randomizzazione permette nel caso medio di avere una complessità di $n \log n$.

countingSort $\Theta(n + k)$, richiede $\mathcal{O}(k)$ memoria aggiuntiva, iterativo. Molto veloce quando $k = \mathcal{O}(n)$.

pigeonholeSort $\Theta(n + k)$, stabile, richiede $\mathcal{O}(n + k)$ memoria aggiuntiva, iterativo. Molto veloce quando $k = \mathcal{O}(n)$.

countingSort e pigeonholeSort vengono utilizzati quando sappiamo qualcosa sui dati in input.

bucketSort $\mathcal{O}(n)$ nel caso i valori siano distribuiti uniformemente, stabile, richiede $\mathcal{O}(n)$ spazio aggiuntivo.

shellSort $\mathcal{O}(n\sqrt{n})$, stabile, adatto per piccoli valori, sequenze quasi ordinate. È una versione migliorata di insertionSort.