Delimitazioni inferiori e superiori di algoritmi e problemi - IntegerSort

Definizione

Un algoritmo A ha complessità (costo di esecuzione) $O(f(n))$ rispetto ad una certa risorsa di calcolo, se la quantità $r(n)$ di risorsa usata da A nel caso peggiore su istanze di dimensione n verifica la relazione $r(n) = O(f(n))$.

Definizione

Un algoritmo A ha complessità (costo di esecuzione) $\Omega(f(n))$ rispetto ad una certa risorsa di calcolo, se la quantità $r(n)$ di risorsa usata da A nel caso peggiore su istanze di dimensione n verifica la relazione $r(n) = \Omega(f(n))$.

Definizione

Un problema P ha una complessità $O(f(n))$ rispetto ad una risorsa di calcolo se esiste un algoritmo che risolve P il cui costo di esecuzione rispetto quella risorsa è $O(f(n))$.

Definizione

Un problema P ha una complessità $\Omega(f(n))$ rispetto ad una risorsa di calcolo se ogni algoritmo che risolve P ha costo di esecuzione nel caso peggiore $\Omega(f(n))$ rispetto quella risorsa.

Ottimalità di un algoritmo:

Dato un problema P con complessità $\Omega(f(n))$ rispetto ad una risorsa di calcolo, un algoritmo che risolve P è (asintoticamente) ottimo se ha costo di esecuzione $O(f(n))$ rispetto a quella risorsa.

Per esempio quando si tratta del problema dell'ordinamento possiamo trovare diversi upper bound e lower bound:

- **Upper bound:** $O(n^2)$ ----> Insertion Sort, Selection Sort, Quick Sort, Bubble Sort.
- **Un Upper bound migliore:** $O(nlog(n))$ ----> Merge Sort, Heap Sort.
- **Lower bound:** $\Omega(n)$ ----> un algoritmo per ordinare deve vedere tutti gli elementi.

Quindi abbiamo un gap tra il nostro upper bound e lower bound.

Lower bound per Algoritmi di ordinamento per confronto

Algoritmi di ordinamento per confronto: Tutti gli algoritmi in cui per ordinare gli elementi vengono effettuate delle operazioni di confronto.

Teorema

Ogni algoritmo basato su confronti che ordina n elementi deve fare nel caso peggiore $\Omega(nlog(n))$ confronti.

Possiamo anche dire che il numero di confronti che un algoritmo esegue è un lower bound al numero di passi elementari che esegue.

Corollario

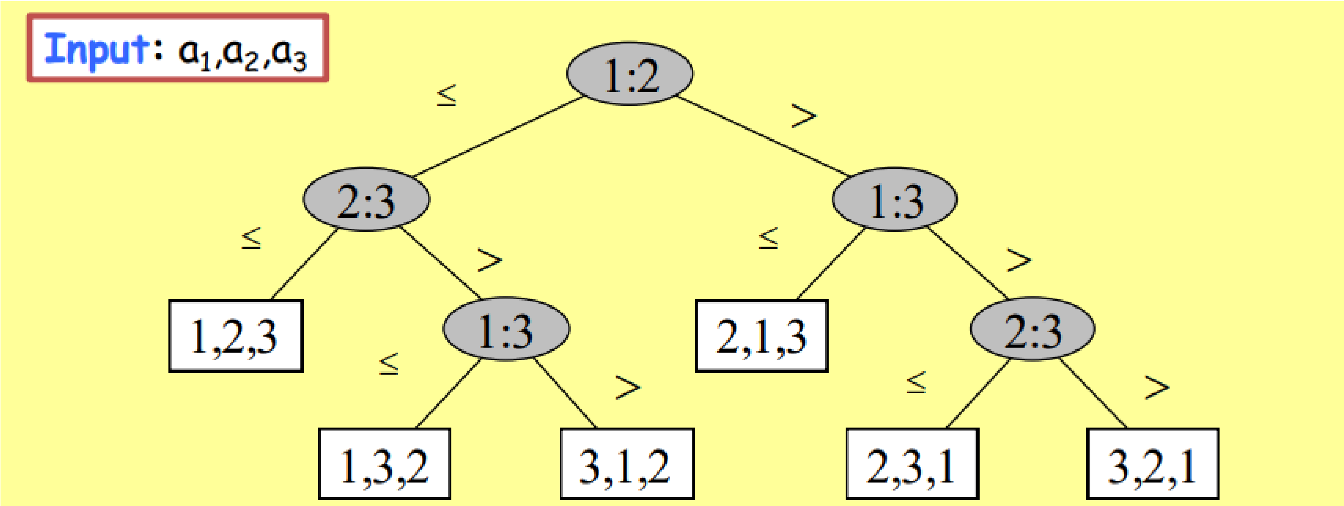
Il Merge Sort e l'Heap Sort sono algoritmi ottimi.

Albero di decisione

Gli algoritmi di confronto possono essere descritti tramite un albero di decisione che **descrive i confronti** che l'algoritmo esegue quando opera su un input di una determinata dimensione. I movimenti dei dati e tutti gli altri aspetti dell'algoritmo sono ignorati.

Quindi descrive le diverse sequenze di confronti che A potrebbe fare su istanze di dimensione n.

- **Nodo interno** (non foglia): $i : j$ confronto tra a_i, a_j.
- **Nodo foglia**: Output dell'algoritmo, nonché permutazione degli oggetti.



⚠ **Osservazioni:**

- L'albero di decisione NON è associato ad un problema.
- L'albero di decisione NON è associato solo ad un algoritmo.
- L'albero di decisione è associato ad un algoritmo e a una dimensione dell'istanza.
- L'albero di decisione descrive le diverse sequenze di confronti che un certo algoritmo può eseguire su istanze di una data dimensione.
- L'albero di decisione è una descrizione alternativa dell'algoritmo (customizzato per istanze di una certa dimensione).

Alcune proprietà:

- Per una particolare istanza, i confronti eseguiti dall'algoritmo su quella istanza rappresentano un **cammino radice-foglie**.
- L'algoritmo esegue un **cammino diverso a seconda dell'istanza**.
- Il numero di confronti nel **caso peggiore è pari all'altezza dell'albero di decisione.**
- Un albero di decisione di un algoritmo corretto che risolve il problema dell'ordinamento di n elementi deve avere necessariamente almeno $n!$ foglie.

Lemma

Un albero binario T con k foglie, ha altezza almeno $log_2(k)$

Dimostrazione per induzione su k:

Passo base: $k = 1$ altezza almeno $log_2(1) = 0$

Passo induttivo: $k > 1$
Prendiamo in considerazione il nodo interno v più vicino alla radice che ha due figli (v deve

necessariamente esistere perché $k > 1$).
v ha almeno un figlio u che è radice si un sottoalbero che ha almeno $k/2$ foglie e $< k$ foglie.
T ha altezza almeno:

$$1 + log_2(k/2) = 1 + log_2(k) - log_2(2) = log_2(k)$$

.

Considerando l'albero di decisione di un qualsiasi algoritmo di ordinamento di n elementi otteniamo che l'altezza, h, dell'albero di decisione è almeno $log_2(n!)$

Formula di Stirling: $n! \approx (2\pi n)^{1/2} * (n/e)^n$.

Quindi:

$$h \geq log_2(n!) > log_2(n/e)^n = nlog_2(n/e) = nlog_2(n) - nlog_2(e) = \Omega(nlog(n))$$

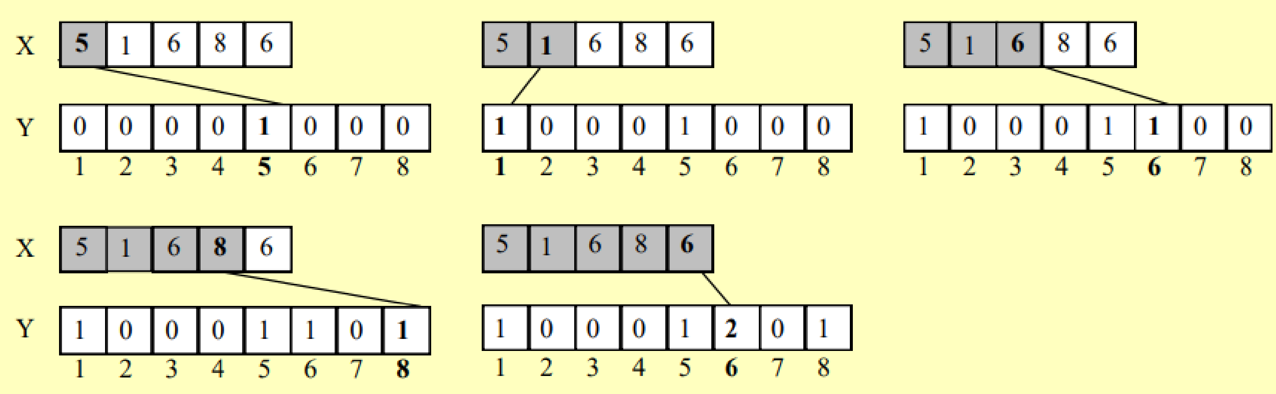
Dove alla seconda disuguaglianza teniamo conto che: $n! > (n/e)^n$.

Algoritmi non basati su confronto: IntegerSort

Un algoritmo basato su confronti non può ordinare n interi piccolo in asintoticamente meno di $nlogn$

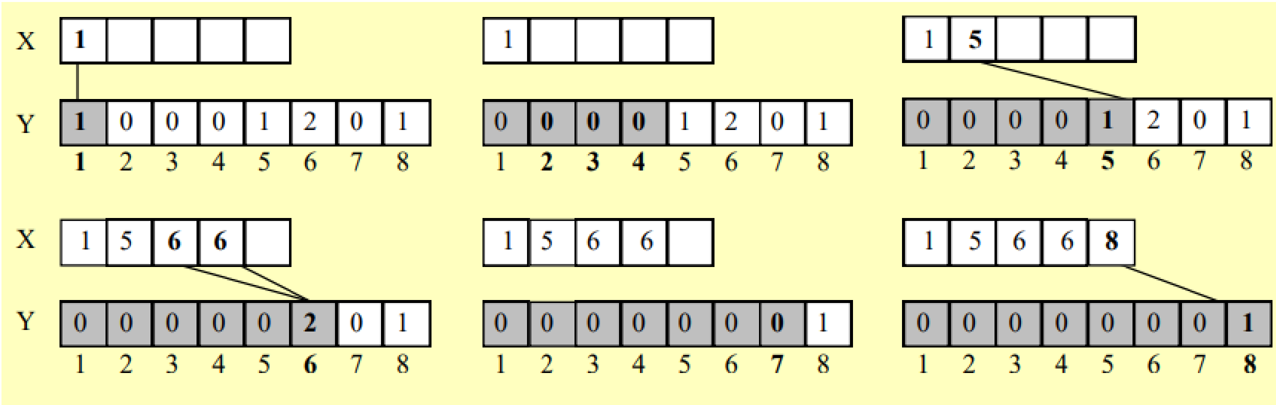
Il primo algoritmo non basato su algoritmi che vediamo è IntegerSort, è diviso in due fasi:

1. Per ordinare n interi con valori da $[1, k]$, manteniamo un array y di k contatori tale che $y[x] =$ numero di volte che il valore x compare in X.



(a) Calcolo di Y

2. Scorriamo y da sinistra verso destra e, se $y[x] = k$, scrive in X il valore x per k volte.



(b) Ricostruzione di X

Codice in python:

```
def IntegerSort(X, k):

    y = [0] * k  # Inizializza un array di k elementi a 0 O(1)

    for i in range(len(X)): # O(n)
        y[X[i]] += 1  # Incrementa il counter array

    j = 0  # Inizializza l'indice per l'array ordinato


    for i in range(k): # O(k)
        while y[i] > 0: # O(n+k)
            X[j] = i
            j += 1
            y[i] -= 1
    return X
```

Possiamo calcolare la complessità facendo il calcolo della seguente sommatoria:

$$\sum_{i=1}^{k}(1 + y[i]) = \sum_{i=1}^{k} 1 + \sum_{i=1}^{k} y[i] = k + n$$

Analisi:

- Tempo $O(1) + O(k) = O(k)$ per inizializzare y a 0.
- Tempo $O(1) + O(n) = O(n)$ per calcolare i valori dei contatori.
- Tempo $O(k + n)$ per ricostruire X

Questo algoritmo è molto veloce per interi piccoli, infatti per n abbastanza piccolo la complessità è lineare, cioè $k = O(n)$, il che non contraddice il lower bound $\Omega(nlog(n))$ perché

IntegerSort non è un algoritmo di ordinamento per confronto.