Il Problema dell'Ordinamento 1/2

Algoritmi 1 – Lezione 7

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 1

Indice

Il problema dell'Ordinamento

- L'ordinamento è un problema fondamentale in informatica. Consiste nel riordinare gli elementi di un insieme secondo un criterio (es. crescente o decrescente).

- Gli algoritmi di ordinamento sono essenziali in molte applicazioni: migliorano l'efficienza delle ricerche, ottimizzano l'uso delle risorse e facilitano la soluzione di problemi complessi.

In queste slide presenteremo alcuni algoritmi di ordinamento semplici e concluderemo con un limite inferiore (lower bound) sul tempo necessario all'ordinamento.

Ordinamento per Selezione (Selection Sort)

- All'inizio l'array è diviso in due parti: parte ordinata (vuota) e parte non ordinata (tutti gli elementi).

L'algoritmo costruisce gradualmente l'array ordinato inserendo un elemento alla volta nella posizione corretta all'interno della parte già ordinata.

- Ad ogni passo, si cerca il minimo nella porzione non ordinata e lo si scambia con il primo elemento non ordinato.
- Quell'elemento diventa parte della porzione ordinata.

Codice Python: Selection Sort

Ecco una semplice implementazione dell'algoritmo in Python:

```
def SelectionSort(A):
    n = len(A)
    for i in range(n - 1):
        # Trova il minimo nella parte non ordinata
        indice_min = i
        for j in range(i + 1, n):
            if A[j] < A[indice_min]:
                indice_min = j
        # Scambia il minimo con il primo elemento non ordinato
        A[i], A[indice_min] = A[indice_min], A[i]
```

Complessità e caratteristiche di Selection Sort

Complessità:

- Il ciclo esterno compie $n-1$ iterazioni.
- Per ogni i, il ciclo interno esegue $n-i-1$ confronti.
- La complessità totale è:

$$T(n) = \sum_{i=0}^{n-2}(n-i-1)\cdot\Theta(1) = \sum_{j=1}^{n-1} j\cdot\Theta(1) = \Theta(n^2)$$

Punti di forza:

- Algoritmo **in-place**, non richiede memoria aggiuntiva.

Punti di debolezza:

- Inefficiente su grandi dataset: sempre $\Theta(n^2)$, anche se l'array è già ordinato.

Ordinamento per Inserimento (Insertion Sort)

- L'array è diviso in parte ordinata e parte non ordinata. All'inizio la parte ordinata contiene solo il primo elemento.

L'algoritmo costruisce gradualmente l'array ordinato inserendo un elemento alla volta nella posizione corretta all'interno della parte già ordinata.

- Si prende il primo elemento della parte non ordinata e lo si confronta con quelli della parte ordinata.
- Gli elementi maggiori si spostano a destra per creare lo spazio per l'inserimento.

Codice Python: Insertion Sort

Ecco una possibile implementazione dell'algoritmo in Python:

```
def InsertionSort(A):
    n = len(A)
    for i in range(1, n):
        x = A[i]
        # x è l'elemento da inserire nella parte ordinata
        j = i - 1
        # Sposta a destra gli elementi maggiori di x
        while j >= 0 and A[j] > x:
            A[j + 1] = A[j]
            j -= 1
        # Inserisce x nella posizione corretta
        A[j + 1] = x
```

Complessità dell'Insertion Sort

Struttura: il ciclo *for* esegue $n-1$ iterazioni, e al passo i il `while` può compiere fino a $O(i)$ confronti.

Caso migliore:
L'array è già ordinato. Il `while` non esegue alcuno spostamento, quindi:

$$T(n) = \sum_{i=1}^{n-1} \Theta(1) = \Theta(n)$$

Caso peggiore:
L'array è ordinato in ordine decrescente. Ogni elemento della parte ordinata deve essere spostato:

$$T(n) = \sum_{i=1}^{n-1} \Theta(i) = \Theta(n^2)$$

Nota: il caso ottimo e quello pessimo hanno complessità asintoticamente diverse. Vale la pena di considerare il caso medio.

Caso medio dell'Insertion Sort

Nel caso medio, ogni elemento deve essere inserito in una posizione casuale nella parte ordinata.

- In media, ogni elemento viene spostato a metà della sua distanza si verificheranno dunque circa $\frac{i}{2}$ spostamenti.
- Il numero totale atteso di spostamenti è:

$$E = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} \sum_{i=1}^{n-1} i = \frac{n(n-1)}{4}$$

- Pertanto, nel caso medio:

$$T(n) = \Theta(n^2)$$

Osservazione: sebbene in scenari favorevoli l'algoritmo possa comportarsi meglio, l'ordine di grandezza rimane quadratico nel caso medio.

Ordinamento a Bolle (Bubble Sort)

- L'array è concettualmente diviso in due parti:
  - una parte **ordinata** a destra (all'inizio vuota),
  - una parte **non ordinata** a sinistra (all'inizio contiene tutti gli elementi).

L'algoritmo costruisce gradualmente l'array ordinato facendo "risalire" gli elementi più grandi verso la fine dell'array attraverso scambi tra elementi adiacenti.

- Si confrontano due elementi adiacenti della parte non ordinata.
- Se sono fuori ordine, vengono scambiati.
- Alla fine di ogni passata, l'elemento più grande tra quelli non ancora ordinati si trova nella posizione corretta e viene considerato parte della porzione ordinata a destra.

Dopo i passate, gli ultimi i elementi dell'array sono già in posizione corretta.

Codice base del Bubble Sort

Ecco una possibile implementazione dell'algoritmo in Python:

```
def BubbleSort(A):
    n = len(A)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
```

Analisi della complessità:

$$T(n) = \sum_{i=0}^{n-2}(n-i-1) = \sum_{j=1}^{n-1} j = \Theta(n^2)$$

L'algoritmo ha complessità quadratica anche se l'array è già ordinato: non è ottimizzato.

Bubble Sort ottimizzato

Possiamo ottimizzare l'algoritmo con un flag `scambi`, che interrompe l'esecuzione se l'array è già ordinato:

```
def BubbleSort(A):
    n = len(A)
    for i in range(n - 1):
        scambi = False
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
                scambi = True
        if not scambi:
            return
```

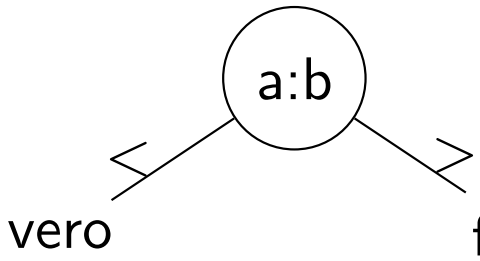
Caso migliore: array già ordinato nessuno scambio complessità $O(n)$.
Caso peggiore: array in ordine decrescente tutti gli scambi $\Theta(n^2)$.

Introduzione al limite inferiore

- Come determinare un **limite inferiore nel caso peggiore** per il costo computazionale degli algoritmi di ordinamento basati su confronti?
- Lo strumento chiave è l'**albero di decisione**, che rappresenta tutte le possibili sequenze di confronti effettuate dall'algoritmo.
- L'idea fondamentale: ogni algoritmo deve essere in grado di distinguere tutte le permutazioni possibili dell'input attraverso confronti tra elementi.

Cos'è un albero di decisione

- Un **albero di decisione** rappresenta il processo decisionale di un algoritmo basato su confronti.
- Considera solo i confronti tra coppie di elementi, ignorando tutte le altre operazioni.
- Se tutti gli elementi sono distinti, ogni confronto tra due elementi ($a < b$) ha due possibili esiti: vero o falso.
- Ogni confronto è un **nodo interno** dellalbero, mentre le **foglie** rappresentano la permutazione finale dellinput.
- La r**adice** rappresenta il primo confronto; ogni cammino dalla radice a una foglia corrisponde a una sequenza di confronti che produce una permutazione.

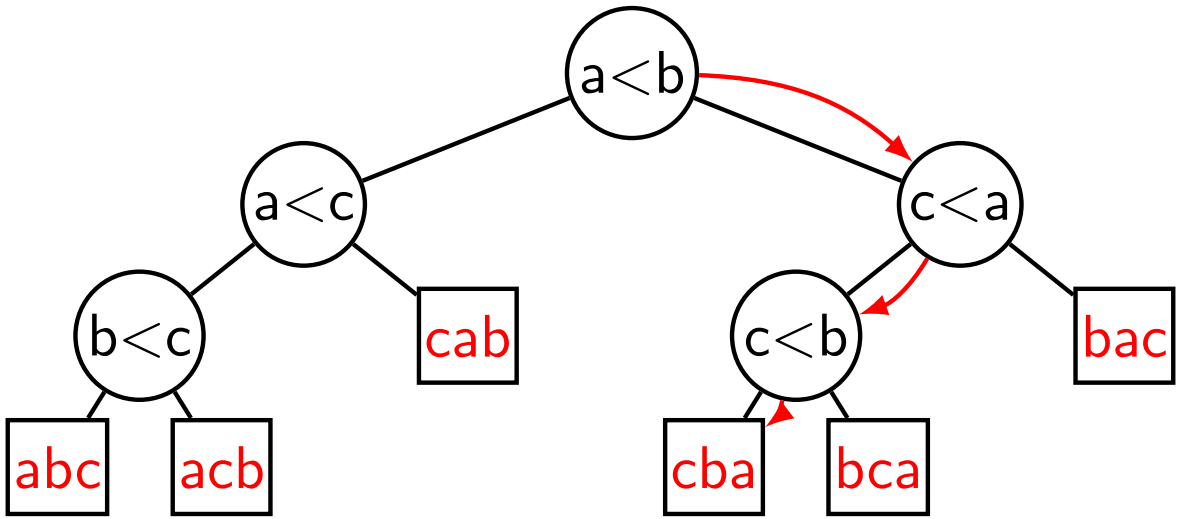


permutazione ordinata dell'input

Figura: Nodo interno (a sinistra) e nodo foglia (a destra) dellalbero di decisione.

Esempio: albero di decisione per Insertion Sort con 3 elementi

- L'albero rappresenta tutte le sequenze di confronti per ordinare $[a, b, c]$.
- Eseguire l'algoritmo significa percorrere un cammino dalla radice fino a una foglia, determinato dagli esiti dei confronti.
- La lunghezza del cammino corrisponde al numero di confronti effettuati per quella permutazione.
- Il cammino più lungo rappresenta il caso peggiore.



Relazione tra il numero di foglie e l'altezza dell'albero

Sia d l'altezza dell'albero di decisione per n elementi.

- Un albero binario di altezza d può avere al massimo 2^d foglie.
- Per ordinare n elementi, servono almeno $n!$ foglie (una per ogni permutazione dellinput).

Da ciò segue:

$$n! \leq 2^d \implies d \geq \lceil \log_2 n! \rceil$$

Poiché d rappresenta il numero minimo di confronti nel caso peggiore:

$$T(n) \geq \log_2(n!)$$

Usando la disuguaglianza $n! \geq (n/2)^{n/2}$:

$$T(n) \geq \log_2 \left(\left(\frac{n}{2} \right)^{n/2} \right) = \frac{n}{2} \log_2 \frac{n}{2} = \Omega(n \log n)$$

Conclusione

- La profondità minima di un albero di decisione per l'ordinamento di n elementi è $\Omega(n \log n)$.
- Quindi, **qualsiasi algoritmo di ordinamento basato su confronti richiede almeno** $\Omega(n \log n)$ **confronti nel caso peggiore**.
- Questo rappresenta il **limite inferiore asintotico** per tutti gli algoritmi di ordinamento basati su confronti.

Algoritmi di ordinamento a complessità lineare

- Abbiamo appena visto che gli algoritmi di ordinamento basati sui confronti hanno, nel caso peggiore, complessità temporale $\Omega(n \log n)$.
- Tuttavia, esistono algoritmi che, in situazioni particolari, ordinano in tempo lineare, ovvero con complessità $O(n)$.
- Questi algoritmi non si basano su confronti diretti tra gli elementi, ma sfruttano proprietà particolari dell'input, come ad esempio un **dominio limitato dei valori**.
- Esempi classici di questa categoria sono:
  - **Counting Sort**
  - **Bucket Sort**
- Nei prossimi lucidi vedremo come funzionano questi algoritmi e in quali situazioni possono garantire effettivamente una complessità lineare.

Counting Sort

- **Counting Sort** è un algoritmo di ordinamento **non basato sui confronti**, adatto quando gli elementi sono **interi non negativi**.
- L'idea principale è **contare le occorrenze** di ciascun valore nell'input per determinare direttamente la posizione finale degli elementi ordinati.
- Procede in due fasi principali:
  - Creazione di un array ausiliario C che conta le occorrenze di ogni valore dell'input.
  - Lettura dell'array C per ricostruire l'array ordinato.
- L'algoritmo non effettua confronti tra elementi e risulta particolarmente efficiente quando il valore massimo dell'input è limitato.

Counting Sort Implementazione Python

```
def Counting_Sort(A):
    k = max(A)
    n = len(A)
    # Crea array C per contare le occorrenze
    C = [0] * (k + 1)
    # Conta quante volte compare ogni elemento
    for i in range(n):
        C[A[i]] += 1
    # Ricostruisce A in ordine crescente
    j = 0
    for i in range(len(C)):
        for _ in range(C[i]):
            A[j] = i
            j += 1
```

Counting Sort Complessità

- La complessità temporale di Counting Sort è:

$$O(n + k)$$

  dove:
  - n è la dimensione dell'array in input,
  - k è il valore massimo presente nell'array.
- Analisi dettagliata dei costi:
  - Calcolare $k = \max(A)$ richiede $O(n)$ iterazioni.
  - Inizializzare l'array dei contatori C richiede $O(k)$.
  - Il primo ciclo *for* per contare le occorrenze esegue n iterazioni.
  - Il secondo ciclo doppio per ricostruire l'array ordinato esegue complessivamente $n + k$ assegnazioni.
- **Caso favorevole:** se $k = O(n)$, la complessità totale diventa $O(n)$.
- **Limite:** se $k \gg n$, l'algoritmo richiede molto spazio ($O(k)$) e tempo, perdendo efficienza.
- **Conclusione:** Counting Sort è adatto quando i valori sono interi non negativi e il loro intervallo è limitato.

Bucket Sort

Il **Bucket Sort** sfrutta la distribuzione dei dati per ordinare gli elementi.

1. L'algoritmo suddivide l'intervallo dei valori in k sotto-intervalli detti *bucket*.
2. Ogni elemento viene **mappato** nel bucket corretto in base al suo valore.
3. Ogni bucket viene ordinato separatamente, tipicamente con un algoritmo semplice come l'**Insertion Sort**.
4. Infine, i bucket vengono **concatenati** per ottenere l'array ordinato.

Come calcolare il bucket di un elemento x con $0 \leq x \leq M$:

$$i = \left\lfloor \frac{x \cdot k}{M+1} \right\rfloor$$

dove k è il numero totale di bucket. La funzione $\lfloor \cdot \rfloor$ assicura che i sia un intero tra 0 e $k-1$.

Bucket Sort: implementazione in Python

```
def Bucket_Sort(A, k):
    # Crea k bucket vuoti
    buckets = [[] for _ in range(k)]
    M = max(A)  # Valore massimo
    # Inserimento degli elementi nei bucket
    for x in A:
        # calcola l'indice del bucket di x
        i = x * k // (M + 1)
        buckets[i].append(x)
    # Ordina i bucket separatamente
    for bucket in buckets:
        bucket.sort()
    # Concatenazione dei bucket
    B = []
    for bucket in buckets:
        B.extend(bucket)
    return B
```

Bucket Sort: complessità struttura generale

La complessità totale di Bucket Sort può essere espressa come:

$$O\left(n + \sum_{i=0}^{k-1} \Theta(\text{costo ordinamento } B[i])\right)$$

- n è il numero di elementi in input.
- $B[i]$ è il bucket i contenente alcuni elementi dell'input.
- La somma rappresenta il costo di ordinare tutti i bucket.

Assumiamo di scegliere il numero di bucket proporzionale alla dimensione dell'input:

$$k = \Theta(n)$$

Questo è fondamentale per ottenere complessità lineare nei casi favorevoli.

Bucket Sort: casi ottimo e pessimo

- **Caso ottimo:** gli elementi sono equamente distribuiti nei bucket.
  - Ogni bucket contiene $\Theta(1)$ elementi.
  - Ordinare ciascun bucket costa $\Theta(1)$, quindi la complessità totale è $O(n)$.
- **Caso pessimo:** tutti gli elementi finiscono in un unico bucket.
  - La complessità dipende dall'algoritmo scelto per ordinarlo (es. $O(n \log n)$ se si usa quicksort o merge sort).
- **Condizioni per la linearità:**
  - Gli elementi devono essere equidistribuiti tra i bucket.
  - Il numero di bucket deve essere proporzionale a n ($k = \Theta(n)$).
- **Conclusione:** Bucket Sort può raggiungere complessità **lineare** $O(n)$, ma solo in condizioni ideali.

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Esercizi

1. Qual è il costo computazionale dei tre algoritmi visti in questa lezione quando la lista da ordinare ha gli elementi che sono tutti uguali? E quando è già ordinata? E quando è ordinata in modo decrescente?
2. Nell'algoritmo di *Insertion sort*, è possibile ricercare la posizione in cui inserire l'elemento i-esimo tramite la ricerca binaria. Come cambia il costo computazionale dell'algoritmo?
3. Modificare il codice proposto per il *Counting sort* in modo che possa essere applicato a interi anche negativi, e affinché la complessità risulti dell'ordine $O(n+k)$, dove n è la dimensione dell'array da ordinare e k è la differenza tra il massimo ed il minimo valore.
4. Un algoritmo di ordinamento si dice **stabile** se mantiene l'ordine relativo degli elementi con valori uguali nell'input. In altre parole, se due elementi a e b sono uguali secondo il criterio di ordinamento, allora a apparirà prima di b nel risultato finale se nell'input a compariva prima di b.

   La stabilità è particolarmente utile in contesti dove si desidera mantenere informazioni aggiuntive che sono collegate agli oggetti da ordinare. Ad esempio, se si ordina una lista di persone per età, ma si desidera che le persone con la stessa età restino ordinate secondo un altro criterio (ad esempio, per nome), un algoritmo stabile manterrà l'ordine originale per quelle persone che hanno lo stesso valore di età.

   Indicare, tra i seguenti algoritmi di ordinamento, quale di essi è stabile e spiegare il motivo:

   - Bubble Sort
   - Selection Sort
   - Insertion Sort