Corso di laurea in Informatica
Introduzione agli Algoritmi

Il problema dell'ordinamento: algoritmi lineari

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Abbiamo dimostrato che gli algoritmi basati sui confronti hanno una complessità minima nel caso peggiore pari a $\Omega(n \log n)$. Tuttavia, esistono algoritmi di ordinamento che **in particolari condizioni** raggiungono una complessità temporale lineare, ossia $O(n)$. Tra questi,

1. *Counting Sort*
2. *Bucket Sort*

sono esempi classici.

Analizzeremo ora questi due algoritmi, vedremo come funzionano e perché sono considerati algoritmi a complessità lineare.

Counting Sort

Il Counting Sort è un algoritmo di ordinamento non basato sui confronti che presuppone che i valori di input siano interi. Funziona contando il numero di occorrenze di ciascun elemento e utilizzando queste informazioni per costruire l'array ordinato. Il funzionamento può essere descritto come segue:

- Si crea un array di conteggio che registra il numero di occorrenze di ciascun valore presente nell'input.
- Si costruisce l'array ordinato iterando sugli elementi dell'input e posizionandoli nell'array di output basandosi sull'array di conteggio.

Esempio

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

n=12, k=8

C:	0	0	0	0	0	0	0	0	0

1) Scorrendo A conteggiamo in C il numero di occorrenze di ciascun valore

	0	1	2	3	4	5	6	7	8
C:	0	2	1	0	2	1	3	1	2

Nota: $\sum_{i=0}^{k} C[i] = n$

2) Scorrendo C ricopiamo in A ciascun indice di C tante volte quanto è il valore in C di quell'indice

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

Ecco il codice dove si assume che gli interi da ordinare siano non negativi.

```
def  Counting_Sort(A):
    k = max(A)
    n = len(A)
    # crea una  lista C di contatori per registrare il
    # numero  di occorrenze degli elementi in A
    C = [0]*(k+1)
    # calcola le occorrenze degli elementi in A in [0,k]
    for i in range(n):
        C[A[i]] += 1
    # reinserisce in A i suoi elementi in modo ordinato
    j = 0
    for i in range(k+1):
        for _ in range(C[i]):
            A[j] = i
            j += 1
```

```
def  Counting_Sort(A):
    k = max(A)
    n = len(A)
    C = [0]*(k+1)
    for i in range(n):
        C[A[i]] += 1
    j = 0
    for i in range(k):
        for _ in range(C[i]):
            A[j] = i
            j += 1
```

$$T(n) = \Theta(n+k)$$

Complessità:

- Il calcolo di k richiede tempo $\Theta(n)$.
- L'inizializzazione die k contatori richiede $\Theta(k)$.
- Il primo for per il conteggio delle varie occorrenze costa $\Theta(n)$.
- I due for annidati per il reinserimento in A degli elementi al posto giusto richiedono $\Theta(n)$.

La complessità dell'algoritmo è dunque $O(n+k)$, dove n è la dimensione dell'input e k il valore massimo presente nell'input.
NOTA CHE: quando k è limitato, cioè $k = O(n)$, l'algoritmo raggiunge una complessità lineare $O(n)$. Tuttavia, se k cresce significativamente rispetto a n, la complessità non sarà più lineare. Questo rende l'algoritmo adatto soprattutto quando il massimo valore dell'input è limitato.

Bucket sort (con k buckets)

Il Bucket Sort è un algoritmo di ordinamento che suddivide l'input in diversi intervalli (*bucket*) e li ordina individualmente. Il funzionamento può essere descritto come segue:

- Dividere l'intervallo dei valori in k bucket.
- Inserire ciascun elemento dell'input nel bucket corrispondente.
- Ordinare individualmente i bucket utilizzando un altro algoritmo (ad esempio, insertion sort).
- Concatenare i bucket ordinati per ottenere l'array finale.

Nel codice, si assume per semplicità che i valori siano non negativi.

Se l'intervallo $[0, M]$ deve essere suddiviso in k sottointervalli di uguale dimensione, allora ogni sottointervallo avrà una lunghezza pari a:

$$\frac{M+1}{k}$$

Per determinare il bucket i in cui l'elemento x deve essere inserito, posso usare la formula:

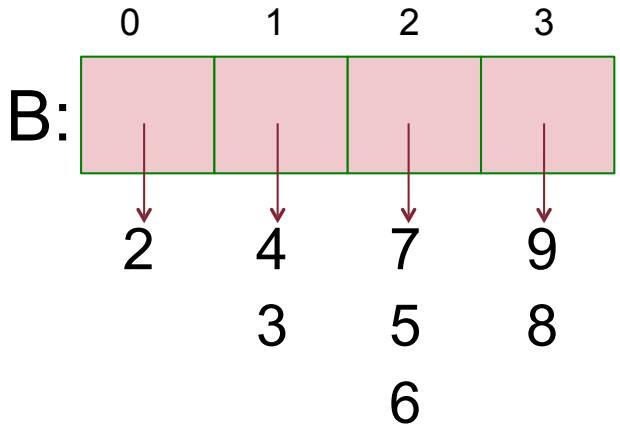
$$i = \left\lfloor \frac{x \cdot k}{M+1} \right\rfloor .$$

La funzione $\lfloor \cdot \rfloor$ assicura che il valore i sia un intero compreso tra 0 e $k-1$.

Esempio Bucket_Sort con $k = 4$

A:	8	9	7	5	6	4	3	2

In questo caso $M = 9$ e l'elemento x finisce nel Bucket $\left\lfloor x \cdot \frac{4}{10} \right\rfloor$



1) Inserimento degli elementi nei bucket
2) Ordinamento di ogni bucket
3) Copiatura dei bucket ordinati

2	3	4	5	6	7	8	9

```python
def Bucket_Sort(A, k):
    # creazione dei secchi vuoti
    buckets = [[] for _ in range(k)]
    #calcolo del valore  massimo in A
    M=max(A)
    # distribuzione degli elementi di A nei k secchi
    for x in A:
        # gli elementi appartengono  all'intervallo [0..M]
        # distribuiamo  gli elementi di A in modo che ogni
        #  secchio riceva  un sottointervallo di  circa M/k
        i = x * k//(M+1)
        buckets[i].append(x)
    # ordinamento dei secchi
    for bucket in buckets:
        bucket.sort()
    # concatenazione dei secchi in un unico array ordinato
    B = []
    for bucket in buckets:
        B.extend(bucket)
    return B
```

```python
def Bucket_Sort(A, k):
    buckets = [[] for _ in range(k)]
    M=max(A)
    # distribuzione degli elementi di A nei k secchi
    for x in A:
        i = x * k//(M+1)
        buckets[i].append(x)
    for bucket in buckets:
        bucket.sort()
    B = []
    for bucket in buckets:
        B.extend(bucket)
    return B
```

Complessità:

1. **creazione dei bucket**: $\Theta(k)$
2. **calcolo del massimo M:** $\Theta(n)$
3. **distribuzione degli elementi dei bucket**: $\Theta(n)$
4. **ordinamento degli elementi nei bucket**:

$$O\left(\sum_{i=0}^{k-1}\Big(\text{costo per ordinare gli elementi del bucket } B[i]\Big)\right)$$

5. **concatenazione degli elementi ordinati dei k buckets**: $O(k+n)$

La complessità è:

$$\Theta\Big(\sum_{i=0}^{k-1}\Big(\text{costo per ordinare gli elementi del bucket } B[i]\Big)\Big)$$

$$\Theta\Big(\sum_{i=0}^{k-1}\Big(\text{costo per ordinare gli elementi del bucket } B[i]\Big)\Big)$$

Assumiamo di prendere un numero di buckets k proporzionale alla dimensione dell'input (non avrebbe senso creare un numero di buckets superiore agli elementi da ordinare), in altre parole $k = \Theta(n)$.

- **Caso pessimo:** tutti gli elementi finiscono nello stesso bucket. In questo caso si ha una complessità O(costo per ordinare n elementi) che dipenderà dall'algoritmo di ordinamento utilizzato. e la complessità dipenderà quindi dall'algoritmo di ordinamento utilizzato.
- **Caso ottimo**: gli elementi sono equidistribuiti nell'intervallo $[0, M]$. In questo caso in ogni bucket finiranno circa $\frac{n}{k} = \Theta(1)$ elementi, ovvero un numero costante di elementi e quindi il costo per ordinare ciascun bucket sarà $\Theta(1)$ indipendentemente dall'algoritmo di ordinamento utilizzato. La complessità dell'algoritmo risulta $O(n)$

Per quanto detto, Per affermare che la complessità di ordinamento del bucket sort è $\Theta(n)$, bisogna fare alcune assunzioni sul comportamento dei dati e sulle operazioni coinvolte nel processo di ordinamento.

1. **Distribuzione uniforme degli elementi** Si assume che gli elementi siano distribuiti uniformemente tra i bucket. Se la distribuzione degli elementi è uniforme, la probabilità che un bucket contenga un numero significativo di elementi è bassa, e quindi ogni bucket conterrà, in media, un numero costante di elementi. In altre parole, il numero di elementi per bucket tende a essere proporzionale a $\frac{n}{k}$, dove n è il numero totale di elementi e k il numero di buckets.

2. **Numero di bucket k proporzionale a n.** Si assume che $k = \Theta(n)$. Se il numero di bucket è troppo piccolo rispetto al numero di elementi, i bucket rischiano di essere troppo affollati, aumentando la complessità di ordinamento.

Numero di Bucket proporzionali ad n e l'equidistribuzione degli elementi tra i bucket assicurano che in ciascun bucket siano presenti $O(1)$ elementi con un costo d'ordinamento costante.

ESERCIZIO risolto (esame dell' appello Marzo 2023).

Data una lista A di n interi non negativi distinti, si vuole determinare se esistono almeno tre numeri tra loro consecutivi di valore inferiore a 100.

Ad esempio se
$A = [101, 5, 9, 31, 33, 10, 100, 4, 8, 32, 500, 11, 99]$,
gli elementi $8, 9$ e 10 così come gli elementi $31, 32$ e 33 rispettano la proprietà mentre $99, 100$ e 101 no.

Progettare un algoritmo che, data la lista A, in tempo $\Theta(n)$ restituisce il valore dell'elemento centrale della terna se questa è presente, -1 altrimenti. Se esistono più terne allora bisogna restituire l'elemento centrale di valore massimo (nell'esempio sopra l'algoritmo dovrebbe restituire 32).

l'algoritmo che consiste nello scorrere l'array per ciascun elemento alla ricerca dei due elementi che possano far parte della terna ha tempo quadratico, $\Theta(n^2)$ mentre ordinare l'array e poi scorrerlo una sola volta alla ricerca di una eventuale terna richiederebbe tempo $\Theta(n \log n)$.

Una prima soluzione corretta consiste nel selezionare i soli elementi di valore al più 100 (questo costerà $\Theta(n)$) ordinarli (questo richiederà tempo costante poiché sono tutti distinti non saranno più di 100) e scorrerli partendo da destra alla ricerca eventuale della prima terna di consecutivi.

Altro possibile algoritmo richiede di creare una lista C di 100 contatori dove in $C[i]$ inserisco il numero di occorrenze di i in A (questa prima parte richiede tempo $\Theta(n)$). Poi scorro i contatori a partire da destra alla ricerca eventuale di 3 locazioni consecutive a valore superiore a 0 (questa seconda parte richiede tempo $\Theta(1)$).

Ecco di seguito possibili codici python dei due algoritmi proposti:

```
def terna1(A):
    B = []
    for x in A:
        if x < 100:
            B.append(x)
    B.sort()
    i = len(B)-2
    while i > 0:
        if B[i]==B[i+1]-1 and B[i]==B[i-1]+1:
            return B[i]
        i -= 1
    return -1
```

```
def terna2(A):
    C = [0]*100
    for i in range (len(A)):
        if A[i] < 100:
            C[ A[i] ] += 1
    i=98
    while i > 0:
        if C[i+1] and C[i] and C[i-1]:
            return i
        i -= 1
    return -1
```

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZI

- Mostrare che il Counting sort è un algoritmo di ordinamento stabile. Cosa accade al tempo di esecuzione del Bucket sort quando tutti gli elementi della lista A sono uguali?

- Generalizzare il Counting sort in modo che funzioni anche quando sono presenti interi negativi.

- Qual è il tempo di esecuzione del Bucket sort nel caso peggiore se come algoritmo di ordinamento usiamo l'Insertion sort?

- Qual è il tempo di esecuzione del Bucket sort nel caso peggiore se come algoritmo di ordinamento usiamo il Merge sort?

- Quale semplice modifica dell'algoritmo di Bucket sort consente di conservare tempo medio lineare e costo $\Theta(n \log n)$ nel caso peggiore?

- Il Bucket sort può essere modificato in modo che l'ordinamento all'interno delle liste sia eseguito tramite Counting sort. Affinché il costo dell'algoritmo sia lineare anche nel caso peggiore, quale ipotesi bisogna fare su k?