Capitolo 2

SORTING: TECNICHE AVANZATE

2.1 Tempo di Calcolo

Per risolvere un problema, spesso sono disponibili molti algoritmi diversi. Come sceglierne il "migliore"?

Un criterio generalmente adottato consiste nel valutare la efficienza di un algoritmo in base alla quantità di risorsa richiesta per il calcolo. La risorsa di interesse dominante è il **tempo** richiesto per eseguire le azioni elementari, ma talvolta è considerato anche lo **spazio** necessario per immagazzinare (o memorizzare) e manipolare i dati.

In pratica, non è ragionevole misurare il *tempo di calcolo* di un algoritmo col numero di secondi richiesto per eseguire su un elaboratore elettronico una procedura che lo descriva. Infatti tale tempo dipende pesantemente dai dati di ingresso, dal linguaggio in cui la procedura è descritta, dalla qualità con cui viene automaticamente tradotta in una sequenza di bit, e dalla natura e velocità dell'elaboratore elettronico.

Occorre una misura "astratta" che tenga maggiormente conto del "metodo di risoluzione" con cui l'algoritmo effettua la computazione, e che offra il grande vantaggio di poter valutare le "prestazioni" di un algoritmo ancora prima di scriverne una implementazione in un qualche linguaggio di programmazione. Poiché i problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati di ingresso, viene spontaneo esprimere il tempo di calcolo come il numero complessivo di *operazioni elementari* (sono elementari le operazioni aritmetiche, logiche, di confronto e di assegnamento) in funzione della dimensione n dei dati di ingresso. Si cerca cioè la funzione $f(n)$ che esprime il numero delle operazioni elementeri eseguite in funzione del numero n dei dati del problema.

Il numero di operazioni è generalmente valutato nel caso peggiore, cioè sul dato d'ingresso più sfavorevole tra tutti quelli di dimensione n, per tenere conto della Legge di Murphy. Talvolta è valutato anche nel caso medio, cioè mediando su tutti i possibili dati di dimensione n, tenendo conto della probabilità con cui possono occorrere.

2.2 Complessità

Nel valutare il *tempo di calcolo* di una procedura, è in genere assai difficile quantificare con esattezza il numero di operazioni elementari eseguite.

Questa difficoltà viene aggirata valutando il numero di operazioni in ordine di grandezza, cioè esprimendolo come limite della funzione al tendere all'infinito della dimensione n, trascurando le costanti moltiplicative ed additive. Si parla così di **complessità asintotica** (o, brevemente, **complessità**) di una procedura. A tal fine, si usa la notazione "**O**" definita come:

> Una funzione g(n) è detta di complessità O(f(n)) se esistono due costanti positive c ed m per cui g(n) $\leq$ c f(n) per ogni n $\geq$ m.

La notazione O fornisce una delimitazione superiore alla complessità, cioè fornisce una valutazione approssimata per eccesso.

Ad esempio la funzione g(n) = $4n^2 + 4n - 1$ è O(n^2), poiché esistono le costanti c = 5 ed m = 4 per cui g(n) $\leq 5n^2$ per $n \geq 4$. Si noti che, in accordo alla definizione, g(n) è O(n^k) per ogni $k \geq 2$, ma in genere si richiede l'ordine più stretto che è appunto O(n^2).

I seguenti ordini di grandezza sono via via crescenti:

- O(1) (ordine costante)
- O($\log n$) (logaritmico)
- O(n) (lineare)
- O($n \log n$) (pseudolineare)
- O(n^2) (quadratico)
- O(n^3) (cubico)
- 0(2^n)
- O($n!$) (fattoriale)
- O(n^n)

A questo punto, può sorgere una domanda legittima: poiché si trascurano le costanti, è sempre lecito preferire, tra due algoritmi, quello avente complessità di ordine più basso? Ad esempio un algoritmo di complessità O(n^2) è da preferire ad un altro di complessità O(n^3) anche se la costante moltiplicativa nel primo è 200 e quella del secondo è 4?

La risposta dipende dalla dimensione attesa dell'input. Sicuramente, per $n < 50$ il secondo algoritmo è più veloce del primo; pertanto se l'algoritmo deve essere eseguito sempre su dati di piccola dimensione si sceglierà quello con tempo O(n^3).

Se invece sono previsti dati di dimensione grande, il rapporto tra i tempi di esecuzione, che è $\frac{4n^3}{200n^2} = \frac{n}{50}$, cresce arbitrariamente con n e l'algoritmo di complessità O(n^2) risulta nettamente il migliore. Poiché in generale l'algoritmo deve poter essere eseguibile su dati di dimensione arbitraria, risulta sensato e conveniente scegliere sempre l'algoritmo la cui complessità è di ordine più basso possibile.

Ricapitolando, per stabilire la complessità di un algoritmo, se ne studia l'ordine di grandezza O del tempo di calcolo, inteso come numero di operazioni elementari eseguite nel caso pessimo (o medio) in funzione della dimensione n dei dati di ingresso.

2.2.1 Complessità di problemi e algoritmi ottimi

Per risolvere un problema, si cerca di scoprire algoritmi di complessità sempre più bassa. Fino a qual punto questa ricerca può essere spinta? In altri termini, esiste un limite inferiore alla complessità, che dipende solo dal problema in esame, che stabilisca uma soglia invalicabile alla complessità di ogni algoritmo, anche non noti, che risolva il problema?

La risposta è sì. Per dimostrare limiti inferiori, si introduce, analogamente all'ordine O, l'ordine di grandezza "Ω":

> Una funzione g(n) è detta di complessità Ω(f(n)) se esistono due costanti positive c ed m per cui nel caso peggiore di input si abbia g(n) $\geq$ c f(n) per ogni n $\geq$ m.

La notazione Ω fornisce una delimitazione inferiore alla complessità di un algoritmo

Se si riesce a dimostrare che qualunque algoritmo per il problema in esame deve avere complessità Ω(f(n)) e se ne trova uno di complessità Ω(f(n)), allora si dice che tale **algoritmo è ottimo**, perché la sua complessità è, in ordine di grandezza, la migliore possibile.

In generale, trovare limiti inferiori significativi è molto più difficile che trovare algoritmi efficienti. Questo perché sono noti pochissimi metodi generali di dimostrazione, per lo più poco potenti.

Due molto semplici sono i seguenti:

1. **Dimensione dei dati**: se un problema ha in ingresso n dati e richiede di esaminarli tutti, allora un lirnite inferiore è $\Omega(n)$.

2. **Eventi contabili**: se un problema richiede che un certo evento sia ripetuto almeno n volte, allora un limite inferiore è $\Omega(n)$.

2.3 Ordinamento

Uno dei problemi classici della programmazione è quello di ordinare in maniera crescente (o decrescente) un insieme di dati. Sono stati sviluppati negli anni svariati algoritmi che risolvono questo problema. L'obiettivo è chiaramente quello di impiegare il minor tempo possibile per completare questa operazione. Di seguito vedremo alcune implementazioni di questi algoritmi, senza la pretesa di dare la miglior implementazione possibile.

Il problema dell'ordinamento di una sequenza arbitraria di numeri richiede di esaminare tutti gli n elementi dell'array. Quindi tale problema ha complessità $\Omega(n)$. Si può dimostrare che il problema dell'ordinamento con input generico ha un limite inferiore $\Omega(n \log n)$.

2.4 Merge Sort

2.4.1 Divide et Impera

Una delle tecniche più importanti per progettare algoritmi efficienti è quella basata sul concetto del **Divide et Impera**:

> Per regnare, occorre tenere distinti i nemici e trarne vantaggio.

L'applicazione di questa massima ad un problema di programmazione consiste nel suddividere il problema in alcuni sottoproblemi più piccoli dello stesso tipo, risolverli e ricombinare (con poco sforzo) le soluzioni parziali per ottenere la soluzione del problema originale.

Il vantaggio degli algoritmi di ordinamento basati sul principio del *divide et impera*, consiste in una maggiore efficienza, dovuta ad un tempo di esecuzione minore rispetto a quello di algoritmi di ordinamento basati su un **approccio incrementale** quali *selection sort* e *insertion sort*.

Nella prossima sezione vediamo un'applicazione del paradigma.

2.4.2 MergeSort

L'algoritmo **MergeSort** applicato ad un array di n elementi consiste nel dividere l'insieme di dati in due sequenze di $n/2$ elementi ciascuna, ordinare queste sequenze e fondere insieme le due metà ordinate in un'unica sequenza ordinata.

In Fig. 3.1 è visualizzato il partizionamento dei dati su un array di otto elementi.

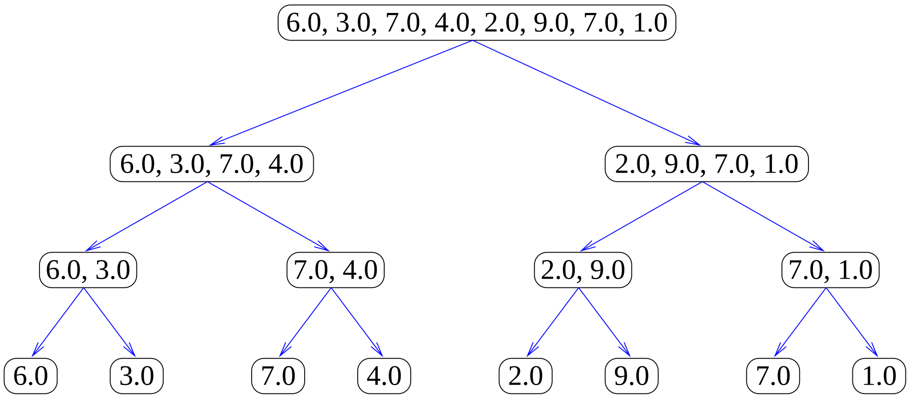

Figura 2.1: Algoritmo MergeSort: albero di partizionamento dei dati

Il partizionamento si arresterà nel momento in cui l'insieme di dati è costituito da un solo elemento. A questo punto si risolve il prblema banale di ordinare un solo elemento e successivamente si fondono sequenze già ordinate per ottenere sequenze più lunghe. In Fig. 3.2 è visualizzata l'operazione di fusione delle varie sequenze.

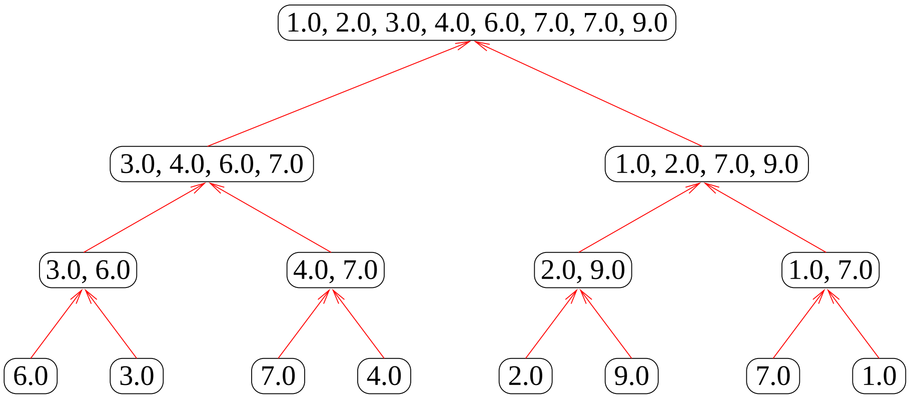

Figura 2.2: Algoritmo MergeSort: fusione dei dati

Di seguito riportiamo una possibile implementazione dell'algoritmo.

```
/*
 * Ordiniamo un array utilizzando la funzione MergeSort.
```

```
 */
#include <iostream.h>

void MergeSort(int d[], int s);
void Merge(double v[], int mezzo, int lung);

void main()
{
  int i;

  int a[] = {8.0, 7.0, 6.0, 4.0, 5.0, 3.0, 2.0, 1.0};
  int b[] = {6.0, 3.0, 7.0, 4.0, 2.0, 9.0, 7.0, 1.0};

  MergeSort(a, 8);

  for (i = 0; i < 8; i++)
    cout << a[i]);

  MergeSort(b, 8);

  for (i = 0; i < 8; i++)
    cout << b[i]);
}

/* Dato un array d di s, ordina l'array in maniera crescente. */
void MergeSort(int d[], int s)
{
  int c = s / 2;                      /* Il punto in cui dividere l'insieme di dati */

  if (s > 1) {
    MergeSort(d, c); /* La prima metà */
    MergeSort(d + c, s - c); /* La seconda metà */
    Merge(d, c, s); /* Fondiamo le sequenze */
  }
}

/* Dato un array v di lunghezza lung diviso in due parti (la prima di lunghezza
 * mezzo la seconda di lunghezza lung - mezzo) ciascuna delle quali ordinata,
 * questa funzione riordina la sequenza fondendo insieme le due parti. */
void Merge(int v[], int mezzo, int lung)
{
  int i, j, k;

  int *b;                         /* Abbiamo bisogno di un vettore di supporto */
  b = new int[lung]

  k = i = 0;
  j = mezzo;
  /* Scorro le due parti di sequenza */
  while ((i < mezzo) && (j < lung)) {
    if (v[i] < v[j])                /* Se è maggiore l'elemento che si trova nella */
      b[k] = v[i++];              /* seconda metà copio quello nella prima metà */
    else                                                              /* o viceversa. */
      b[k] = v[j++];
```

```
    k++;
  }
  /* Potremmo non aver finito di scorrere le due metà. Se non ho finito con
     la seconda metà i dati sono già al loro posto. */
  if (i < mezzo)
    for (j = 0; j < mezzo - i; j++)                /* Copio i dati della prima metà */
      v[k + j] = v[i + j]; /* nel posto giusto */

  /* Ricopio i dati dal vettore di supporto nel vettore originale */
  for (j = 0; j < k; j++)
    v[j] = b[j];

  delete[ ]b;
}
```

Come si può notare la funzione `MergeSort` non fa praticamente nulla: chiama se stessa sulle due metà dell'insieme di dati e poi invoca la funzione `Merge` per ricombinare insieme i dati. Chiaramente se l'insieme di dati è costituito da un solo elemento non c'è niente da fare: l'array è sicuramente già ordinato! Questa costituisce la nostra condiziona d'arresto per le chiamate ricorsive.

La totalità del lavoro è svolto dalla funzione `Merge`. Questa funzione ha necessità di un array di supporto per contenere la sequenza ordinata che si ottiene fondendo le due sottosequenze ordinate, quindi la prima cosa che dovrà fare è quella di allocare lo spazio per questo array, che al più dovrà contenere la somma delle due mezze sequenze.

Alla fine delle operazioni sulle sottosequenze i valori copiati in questo vettore di supporto saranno copiati indietro nell'array originale che risulterà a questo punto ordinato.

Il modo in cui procede la fusione è semplice: si scorrono le due sottosequenze ordinate copiando ogni volta un elemento da una delle due sottosequenze a seconda di dove si trova l'elemento maggiore. Una volta copiato il dato da una delle sottosequenze, si preocederà con la scansione incrementando di uno l'indice della sola sottosequenza da cui abbiamo copiato il dato. Quando abbiamo finito di scorrere una delle due metà si esce da questo ciclo e ci rimane da "sistemare" la parte mancante dell'altra metà.

Poiché alla fine copieremo i dati dall'array di supporto nell'array originale, è inutile copiare nell'array di supporto i dati eventualmente non ancora copiati che appartengono alla seconda sottosequenza: essi si trovano già nella posizione corretta. Dovremo quindi preoccuparci solo dei dati non ancora copiati appartenenti alla prima sottosequenza. Questi dati dovranno andare a finire in fondo all'array originale e possiamo andare a collocarli direttamente nella loro posizione finale senza copiarli nel vettore di supporto.

2.4.3 Costo di Merge Sort

Quando un algoritmo è definito in modo ricorsivo, il suo tempo di esecuzione viene espresso da una relazione di ricorrenza. Consideriamo il caso in cui n sia una potenza esatta di due, nonostante l'algoritmo funzioni in ogni caso, qualunque sia n. Per calcolare $T(n)$, il caso pessimo dell'algoritmo *merge sort* su una sequenza di $n > 1$ elementi [1] , scomponiamo il tempo di esecuzione nel seguente modo:

- il passo di divisione sfrutta un tempo costante $D(n) = \Theta(1)$;
- risolviamo ricorsivamente i due sottoproblemi, ognuno di misura $n/2$, contribuendo di una quantità $2T(n/2)$ al tempo di esecuzione;
- il tempo $C(n)$ impiegato dalla procedura *Merge* per combinare i sottoproblemi contribuisce di una quantità $\Theta(n)$.

Quindi, se $n > 1$ otteniamo:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1) = \\ &= 2T\left(\frac{n}{2}\right) + \Theta(n) \end{aligned}$$

In conclusione

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{se } n = 2^t \end{cases}$$

Utilizzando il metodo fondamentale, per questa relazione di ricorrenza abbiamo $a = 2$, $b = 2$ ed $f(n) = \Theta(n)$ e quindi, applicando il caso 2, si ottiene $T(n) = \Theta(n \log_2 n)$.

Concludiamo quindi che, per i dati in ingresso sufficientemente grandi, l'algoritmo *Merge Sort* risulta essere più efficiente degli algoritmi *Insertion Sort* e *Selection Sort*. Tende però a richiedere una quantità di memoria supplementare proporzionale al numero di elementi da ordinare.

2.5 Counting Sort (Binsort)

Se si assume che ogni elemento di un insieme sia un intero non maggiore di k, è possibile usare l'algoritmo *counting sort*. L'idea che sta alla base di counting sort è che se di un elemento x so quanti elementi sono minori o uguali ad x, allora conosco già quale sarà la sua posizione

[1] Nel caso di sequenze formate da un solo elemento, l'algoritmo impiega un tempo costante.

nell'insieme ordinato. Infatti, se esistono 7 elementi non maggiori di x, ovviamente x andrà all'ottavo posto. Per countig sort abbiamo bisogno, oltre all'array che contiene l'insieme da ordinare, di un array delle stesse dimensioni del primo per contenere i dati ordinati ed un altro ancora di supporto. La dimensione dell'array di supporto deve essere pari a k, il più grande dei numeri del nostro insieme.

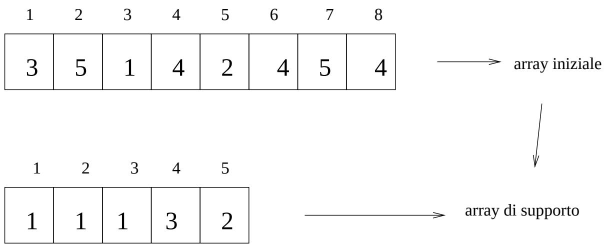

Figura 2.3: La creazione dell'array di supporto di *counting sort* sulla sequenza di elementi $\{3, 5, 1, 4, 2, 4, 5, 4\}$.

Come prima operazione, inseriamo in ogni casella dell'array di supporto la frequenza con cui compare nell'insieme da ordinare il numero pari all'indice della casella. Guardando la Fig.3.3, per esempio, nella posizione 4 dell'array di supporto mettiamo 3, perché il numero 4 ricorre tre volte nell'insieme di partenza. Attenzione che questo va bene per lo pseudocodice, ma in C gli array hanno l'indice che parte da zero!

Seconda operazione (Fig.3.4): nell'array di supporto scriviamo per ogni casella quanti sono gli elementi minori o uguali all'indice della casella. Ciò risulta facile perché si tratta solo di sommare i valori presenti nelle caselle precedenti dell'array.

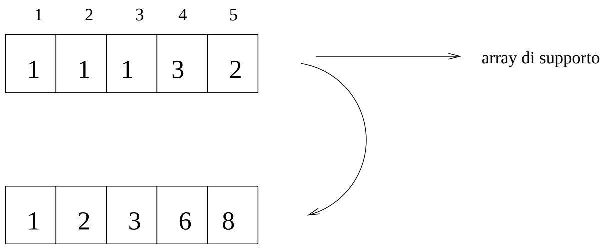

Figura 2.4: Counting sort: seconda operazione.

A questo punto abbiamo quanto serve per cominciare a riempire l'array di arrivo: di ogni elemento dell'insieme di partenza conosciamo posizione (quanti elementi minori o uguali ci sono, dato nella casella dell'array di supporto) e frequenza (valore della csella dell'array di supporto meno valore casella precedente).

Vediamo l'algoritmo in pseudocodice:

```
1.  void function COUNTING (int A[ ], int B[ ], int k)
2.      int k, j, i, S[ ];
3.      for i ← 1 to k
4.          S[i] ← 0;
5.      for j ← 1 to lenghA[ ]
6.          S[A[j] ] ← S[A[j] ]+1;
7.      for i ← 2 to k
8.          S[i] ← S[i]+S[i-1];
9.      for j ← lenght[A] downto 1;
10.         B[S[A[j]] ← A[j];
11.         S[A[j]] ← S[A[j]]-1;
```

Calcolo del tempo richiesto per Counting Sort:

Il ciclo delle linee 3-4, come quello alle linee 7-8, richiede tempo $O(k)$. I cicli alle linee 5-6 e 9-11 richiedono $O(n)$, dove n è la lunghezza dell'array A da ordinare. Tempo totale: $O(k+n)$. Come si vede, un andamento asintotico lineare, che si è ottenuto imponendo alcune limitazioni al tipo di dati in input. L'algoritmo è quindi molto veloce, ma risulta applicabile solo se k è piccolo. Al crescere di k aumenta proporzionalmente la richiesta di memoria e non ci vuole molto a raggiungere il limite della macchina, indipendentemente dalla dimensione dell'input.

2.6 Radix Sort

Si supponga di avere un array i cui elementi sono contraddistinti da un numero a più cifre. Un possibile modo per ordinarli consiste nell'utilizzare un qualsiasi algoritmo di ordinamento considerando solo la cifra meno significativa, quindi ripetere l'ordinamento considerando la seconda cifra meno significativa, e cosi' via. Se l'algoritmo utilizzato ha la proprietà che numeri con lo stesso valore appaiono nell'array di uscita nello stesso ordine di ingresso (proprietà di **stabilità**), avremo alla fine un array ordinato.

Questo è il cosiddetto algoritmo di **Radix Sort**, il cui pseudocodice è semplicissimo:

```
void function RADIX (int A[ ], int d)
    int i;
    for i ← 1 to d
        Ordina A sulla cifra d con un algoritmo stabile;
```

Un buon candidato ad essere utilizzato con Radix Sort è Counting Sort, che è stabile.

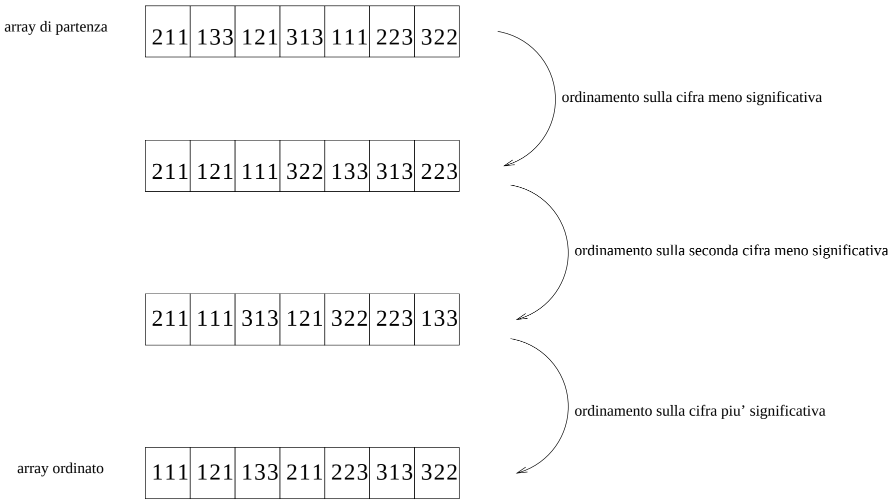

Figura 2.5: Esempio di funzionamento Radix Sort

2.7 Bucket Sort

Se i dati da ordinare sono numeri distribuiti uniformemente nell'intervallo [0,1) si può usare l'algoritmo di ordinamento chiamato **Bucket Sort**. L'idea è di dividere l'intervallo [0,1) in n sottointervalli, dove n è il numero dei dati in input. Quindi distribuire gli n dati associando ciascuno al relativo sottointervallo (detto bucket, secchio, da cui il nome dell'algoritmo). Avendo supposto una distribuzione uniforme su [0,1) ci si aspetta di avere pochi elementi per intervallo. Al limite, per distribuzione perfetta, uno solo. Ora si tratta di ordinare uno alla volta i gruppi di dati in ogni bucket. Quindi leggendo tutti i dati ordinatamente dal primo all'ultimo bucket si ottiene l'ouput richiesto.

Per un input di dimensione n bucket sort richiede un array ausiliario B[n] di liste collegate (Fig.3.7). In pratica gli elementi di B saranno i puntatori sentinella delle liste. Per ordinare i singoli bucket si usa insertion sort.

Di seguito l'algoritmo Bucket Sort in pseudocodice, dove $lungh$=(dimensione di A) e int(x)=(parte intera di x):

void function *BUCKET* (**double** *A*[ ], **int** *lungh*)
  int *i*;
  for *i* ← 1 **to** *lungh*
    inserisci A[*i*] *in B*[*int*(*nA*[*i*])];
  for *i* ← 0 **to** *n-1*
    ordina b[*i*] *con Insertion Sort*;
  concatena ordinatamente B[*1*], *B*[*2*], *B*[*3*], ... , *B*[*n*];

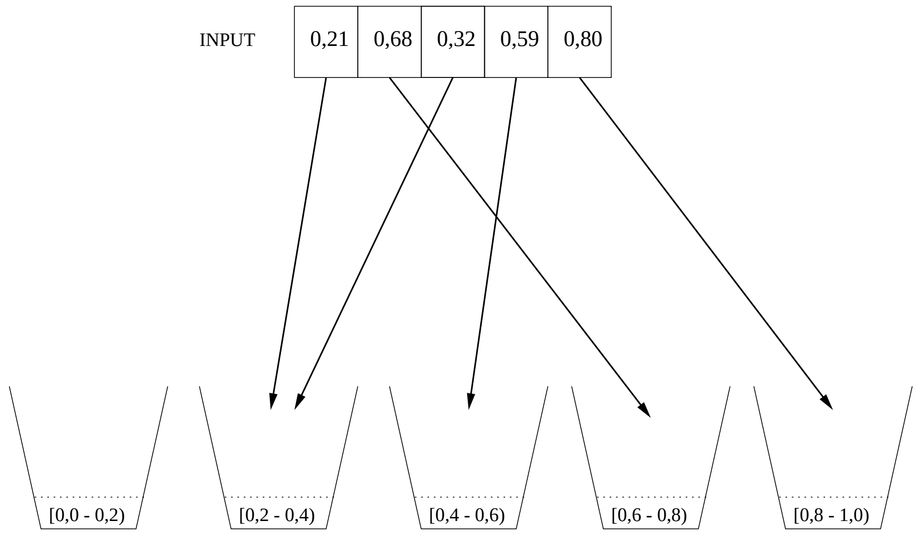

Figura 2.6: Principio di funzionamento di Bucket Sort

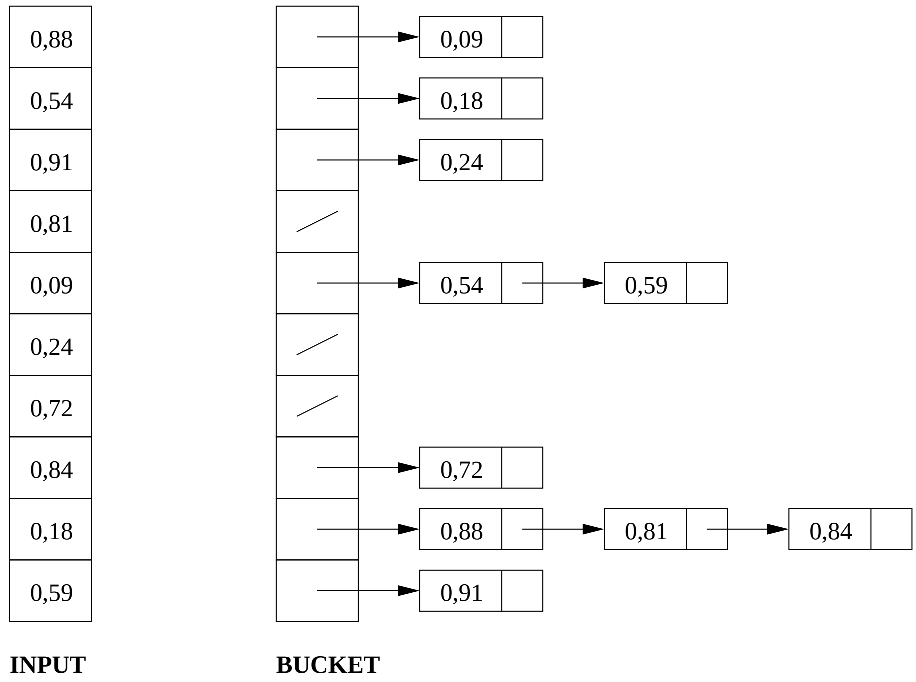

Figura 2.7: Bucket Sort in azione su un input di dieci elementi, dopo l'esecuzione della linea 4 dello pseudocodice.

Bucket sort lavora con complessità lineare nel caso medio. In sostanza è buono fintantoché la distribuzione dei dati su [0,1) è "abbastanza uniforme".