Laboratorio di Informatica
Algoritmi e Strutture Dati
Note

Roberto Battiti   Marco Tomazzoni [1]

19 maggio 2000

Avvertenza: Le note sono in versione provvisoria: lavori in corso!

[1] Si ringrazia Alessandro Villani i cui appunti sono stati ampiamente saccheggiati.

Indice

Capitolo 1

LISTE, PILE E CODE

1.1 Liste collegate

Una semplice struttura dati è la *lista collegata circolare*, una raffigurazione della quale è mostrata in Fig. 1.1 [1]. Una lista collegata è la rappresentazione di un insieme ordinato dinamico (dinamico = che può aumentare o diminuire i suoi membri).

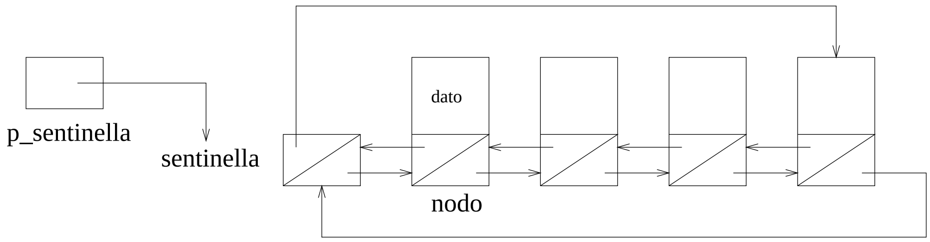


Figura 1.1: Schema di una lista collegata circolare.

Ogni elemento della lista collegata, detto *nodo*, contiene due puntatori, uno al prossimo nodo ed uno al nodo precedente della lista, ed un dato (il dato da caricare nella lista). Spesso il primo nodo della lista è chiamato testa, l'ultimo coda. Un nodo particolare, *sentinella*, contiene due puntatori, uno al primo ed uno all'ultimo nodo della lista, ma non consente di caricare nessun dato. La sentinella sta tra la testa e la coda. La sua utilitá è piú nella chiarezza del codice che nell'incremento delle prestazioni.

Definiamo una classe `Lista` la cui struttura rispecchia quella mostrata in Fig. 1.1 e i cui oggetti rappresentano liste collegate di oggetti

[1]Questo non è l'unico modo per implementare una lista. Vi sono versioni che, ad esempio, non fanno uso del nodo *sentinella*. Esse prevedono che l'ultimo nodo contenga, oltre al puntatore al nodo precedente, il puntatore nullo ed introducono una variabile puntatore, che chiamiamo *testa*, che punta al primo nodo della lista.

di tipo `nodo`. In Fig. 1.2 è mostrato il file di intestazione (header file) per le classi `nodo` e `Lista`.

```
//
// File Lista1.H
// Header File per le liste collegate
//


class nodo {

public:
  nodo* next;    Puntatore al prossimo nodo
  nodo* prev;    Puntatore al nodo precedente
  int data;      Dato contenuto nel nodo
};


class Lista {
  nodo* sentinel;  // Sentinella
  nodo* current;   // Nodo corrente
  int len;         // Lunghezza della lista

public:
  Lista();       // Costruttore
  ~Lista();      // Distruttore

  void Next();     // Sposta il nodo corrente sul nodo successivo
  void Previous(); // Sposta il nodo corrente sul nodo precedente

  void GoToFirst()       // Sposta il nodo corrente sul primo nodo della lista
    { current = sentinel->next; }
  void GoToEndOfList()        // Sposta il nodo corrente oltre l'ultimo nodo
    { current = sentinel; }   // della lista

  int IsEmpty()             // Controllo se la lista e' vuota
    { return (sentinel == sentinel->next); }
  int IsFirst() const       // Controllo se il nodo corrente e' sul primo nodo
    { return (current == sentinel->next); }
  int IsEndOfList() const                 // Controllo se il nodo corrente e'
    { return (current == sentinel); }     // oltre l'ultimo nodo

  void Write(int d);         // Cambia il dato del nodo corrente
  int Read();                // Legge il dato del nodo corrente

  void Insert(int d);        // Inserisce un nodo nella posizione corrente
  int Delete();              // Cancella un nodo dalla posizione corrente

  int GetLength()            // Ritorna la lunghezza della lista
    { return len; }
};
```

Figura 1.2: Header file per la classe Lista.

Come abbiamo detto in precedenza, la classe `nodo` contiene due variabili di istanza `next` e `prev` di tipo `nodo*` (puntatore ad oggetti di tipo `nodo`) e una variabile `data` di tipo `mytype`, il dato che vogliamo caricare nel nodo. La classe Lista è dichiarata friend class in Node Tali variabili di istanza sono dichiarate pubbliche in modo che a tutte le funzioni membro della classe `Lista` sia permesso di accedervi.

La classe `Lista`, come abbiamo già accennato, rappresenta una lista collegata di oggetti di tipo `nodo`. La variabile di istanza più importante è la variabile `sentinel`, puntatore al nodo sentinella. Un'altra variabile di istanza di tipo `nodo*`, chiamata `current`, designa il nodo della lista su cui attualmente si è posizionati.

In Fig. 1.3 è mostrato il file sorgente per la classe `Lista`. Analizziamo in dettaglio le funzioni `Insert(int d)` e `Delete()`:

- la funzione `Insert(int d)` inserisce un nodo contenente il dato `d` immediatamente prima del nodo designato da `current` (in Fig. 1.5 è mostrata una schematizzazione dell'implementazione della funzione). A tale scopo viene creato, con un'istruzione `new`, un oggetto `newnode` di tipo `node` contenente il dato `d`; ai valori dei puntatori `newnode->next` e `newnode->prev` vengono assegnati rispettivamente i valori `current` e `current->prev` (l'indirizzo del nodo che precede `current`). Poi viene aggiornato il valore del `current->prev` (in Fig. 1.5 rappresentato da una freccia tratteggiata) e il valore del puntatore `next` del nodo che precedeva `current` viene posto uguale a `newnode`; dopo tutte queste operazioni di assegnazione ed aggiornamento, il valore di `len` (la lunghezza attuale della lista, ovvero il numero di nodi escluso `sentinel`) viene incrementato di 1;

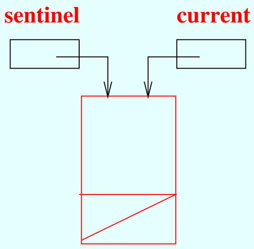
- la funzione `Delete()` rimuove il nodo designato da `current`, operazione permessa solo nel caso in cui `current` non coincida con `sentinel`, e restituisce il dato contenuto nel nodo rimosso. A tale scopo `current->data` viene salvato in una variabile temporanea `tmp` e `current->next` viene salvato in un puntatore temporaneo `tmpp` in modo che, alla fine, `current` sia il nodo che seguiva il vecchio `current`İn seguito vengono aggiornati i valori di `current->next->prev` e di `current->prev->next`, puntatori dei nodi rispettivamente successivo e precedente a `current` (le frecce tratteggiate) e `current` viene distrutto con un'istruzione `delete`.

```
//
// File Lista1.C
// File di implementazione della classe Lista
//

#include <iostream.h>
#include "Lista1.H"

Lista::Lista()
```

Costruttore: crea una lista vuota




```
{
  sentinel = new nodo;

  sentinel->next = sentinel;
  sentinel->prev = sentinel;
  current = sentinel;
  len = 0;
}

Lista::~Lista()
```

Distruttore

```
{
  GoToFirst();
  while (!IsEndOfList())
    Delete();
  delete sentinel;
}

void Lista::Next()
{
  if (IsEndOfList()) {
    cerr << "Errore in Next: sei oltre l'ultimo nodo!"
         << endl;
    exit(-1);
  }
  else
    current = current->next;
}

void Lista::Previous()
{
  if (IsFirst()) {
    cerr << "Errore in Previous: sei sul primo nodo!"
         << endl;
    exit(-1);
  }
  else
    current = current->prev;
}

void Lista::Write(int d)
{
  if (IsEndOfList()) {
    cerr << "Errore in Write: sei oltre l'ultimo nodo!" << endl;
    exit(-1);
  }
  else
    current->data = d;
}
```

Continua...

```
int Lista::Read()
{
  if (IsEndOfList()) {
    cerr << "Errore in Read: sei oltre l'ultimo nodo!" << endl;
    exit(-1);
  }
  else
    return (current->data);
}

void Lista::Insert(int d)
{
  node* newnode = new nodo;

  newnode->next = current;
  newnode->prev = current->prev;
  newnode->data = d;
  current->prev = newnode;
  newnode->prev->next = newnode;
  len++;
}

int Lista::Delete()
{
  if (IsEndOfList()) {
    cerr << "Errore in Delete: sei oltre l'ultimo nodo!" << endl;
    exit(-1);
  }
  int tmp = current->data;
  node* tmpp = current->next;

  current->next->prev = current->prev;
  current->prev->next = current->next;
  delete current;
  len--;
  current = tmpp;
  return tmp;
}
```

Figura 1.3: File di implementazione della classe `Lista`

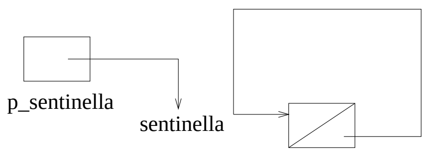


Figura 1.4: Schema di una lista vuota.

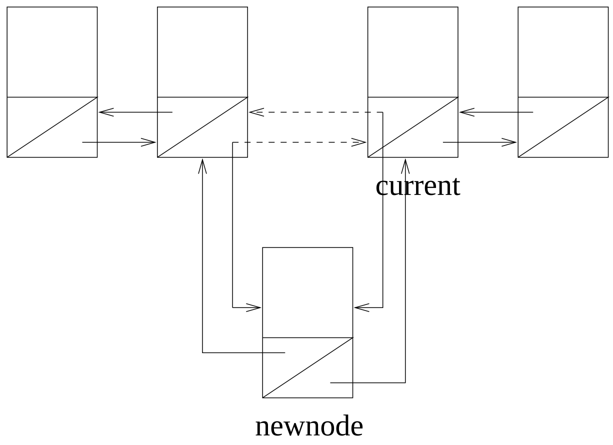


Figura 1.5: La funzione `Insert(int d)`

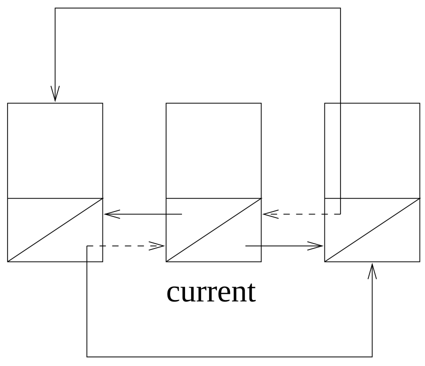


Figura 1.6: La funzione `Remove()`

Capitolo 2

SORTING: TECNICHE AVANZATE

2.1 Merge Sort

2.1.1 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.

2.1.2 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. 2.1 è visualizzato il partizionamento dei dati su un array di otto elementi.

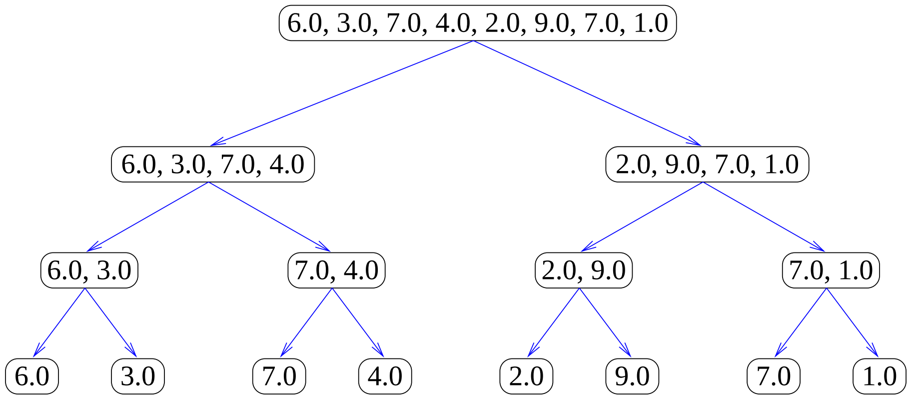


Figura 2.1: Algoritmo MergeSort: albero di partizionamento dei dati

Il partizionamento si arresterà nel momento in cui l'insieme di dati è costituito da un solo elemento. A questo punto si risolve il problema banale di ordinare un solo elemento e successivamente si fondono sequenze già ordinate per ottenere sequenze più lunghe. In Fig. 2.2 è visualizzata l'operazione di fusione delle varie sequenze.

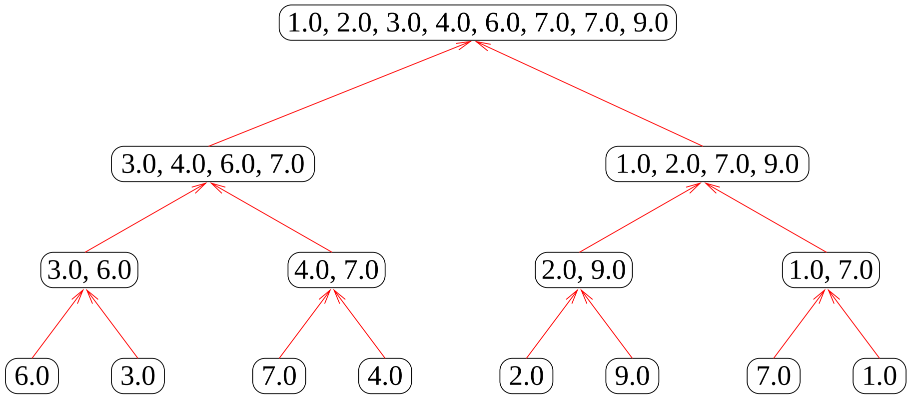


Figura 2.2: Algoritmo MergeSort: fusione dei dati

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);
```

```
int 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.

2.1.3 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.

2.2 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 nell'insieme ordinato. Infatti, se esistono 7 elementi non maggiori di x, ovviamente x andrà all'ottavo posto. Per countig sort abbiamo bisogno,

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

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.

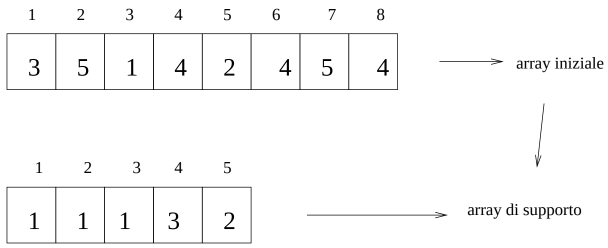


Figura 2.3: La creazione dell'array di supporto di *counting sort* sulla sequenza di elementi $\{3, 5, 1, 4, 2, 4, 5, 4\}$.

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.3.1, 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.2.4): 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.

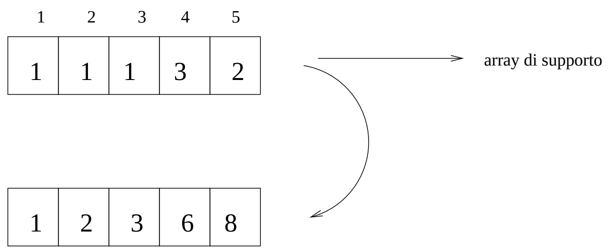


Figura 2.4: Counting sort: seconda operazione.

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:

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 massimo della macchina, indipendentemente dalla dimensione dell'input.

2.3 Radix Sort

Supponiamo 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.

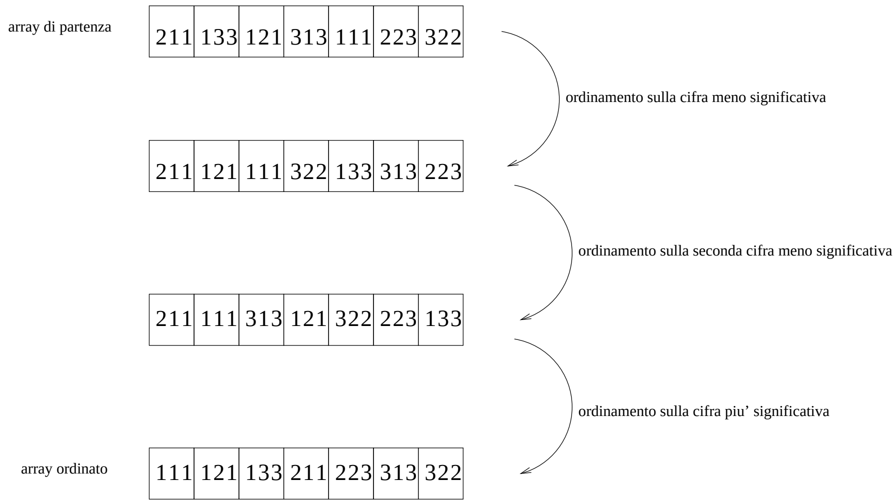


Figura 2.5: Esempio di funzionamento Radix Sort

2.4 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 ingresso. 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'output richiesto.

Per un input di dimensione n bukcet sort richiede un array ausiliario B[n] di liste collegate (Fig.2.6). In pratica gli elementi di B saranno i puntatori sentinella delle liste. Per ordinare i singoli bucket si può usare 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** $i \leftarrow 1$ **to** *lungh*;
  - *inserisci* A[i] *in* b[*int*(nA[i])];
- **for** $i \leftarrow 0$ **to** n-1;
  - *Ordina b*[i] *con Insertion Sort*;
- *concatena ordinatamente* B[1],B[2], ..., B[n];

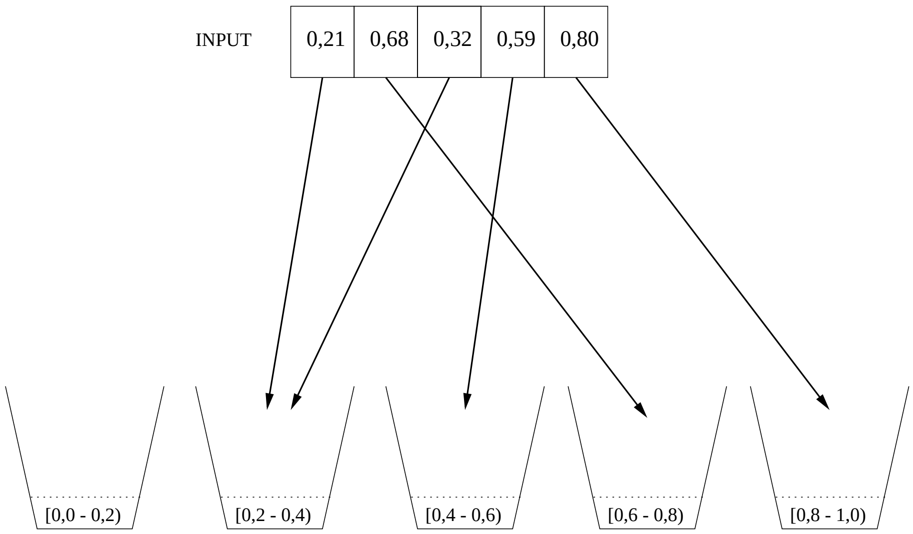


Figura 2.6: Principio di funzionamento Bucket Sort

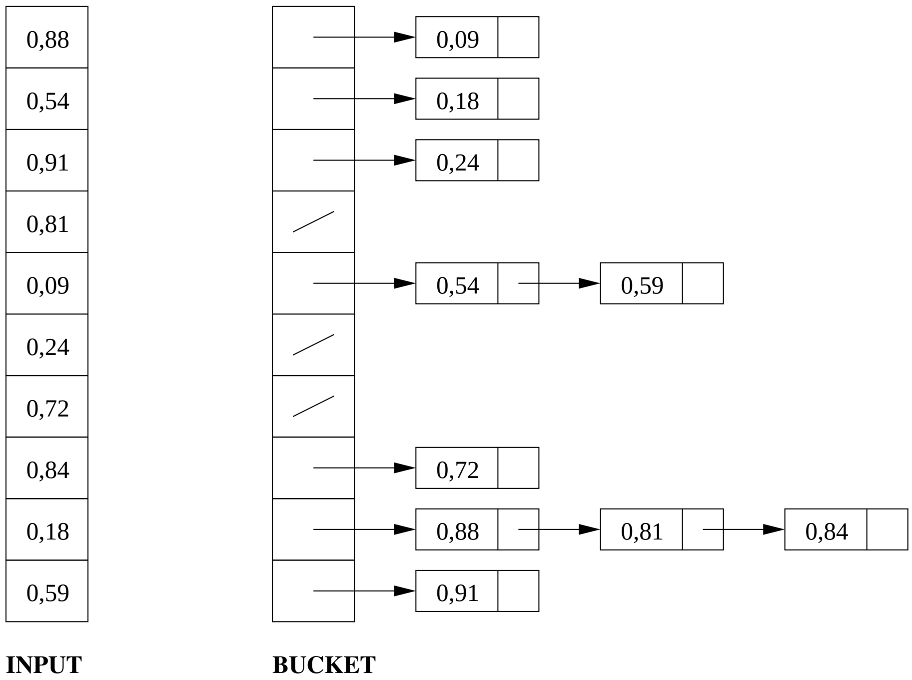


Figura 2.7: Bucket Sort in azione su input di 10 elementi dopo l'esecuzione della riga 4 dello pseudocodice

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

Capitolo 3

POLINOMI ED FFT

3.1 Polinomi

Un polinomio nella variabile x su un campo algebrico F è una funzione $A(x)$ che può essere rappresentata come:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

dove a_j sono i coefficienti ed $n-1$ è il grado di un polinomio (supponiamo che a_{n-1} sia diverso da zero). I coefficienti sono estratti dal campo F, tipicamente l'insieme **C** dei numeri complessi.

La *somma di due polinomi* (per comodità supponiamo che siano dello stesso grado $n-1$) è la funzione $C(x)$ tale che $C(x) = A(x) + B(x)$ per tutti gli x del campo sottostante. $C(x)$ è definita come segue:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$B(x) = \sum_{j=0}^{n-1} b_j x^j$$

$$C(x) = A(x) + B(x) = \sum_{j=0}^{n-1} c_j x^j$$

dove $c_j = a_j + b_j$.

La *moltiplicazione di due polinomi* (per comodità supponiamo che siano dello stesso grado $n-1$) è la funzione $C(x)$ tale che $C(x) = A(x)B(x)$ per tutti gli x del campo sottostante. $C(x)$ è definita come segue:

$$C(x) = A(x)B(x) = \sum_{j=0}^{2n-2} c_j x^j$$

dove:

$$c_j = \sum_{k=0}^{j} a_k b_{j-k}$$

Si noti che il grado del prodotto è dato dalla somma dei gradi dei due polinomi.

3.2 Rappresentazione dei polinomi

3.2.1 Rappresentazione con coefficienti

La rappresentazione con coefficienti di un polinomio

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

è un vettore di coefficienti:

$$(a_0, a_1, ..., a_{n-1})$$

Nelle equazioni matriciali il vettore sarà considerato un vettore colonna.

La rappresentazione con coefficienti è conveniente per **valutare un polinomio**, cioè per trovare il valore $A(x_0)$ a partire da un certo x_0.

La valutazione richiede tempo $\Theta(n)$ se si usa la regola di Horner:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + ... + x_0(a_{n-2} + x_0(a_{n-1}))...))$$

L'addizione di due polinomi rappresentati con coefficienti richiede banalmente tempo $\Theta(n)$.

Purtroppo, la **moltiplicazione di due polinomi** rappresentati con coefficienti richiede tempo $\Theta(n^2)$ se si considera il metodo descritto in precedenza (ogni coefficiente a deve essere moltiplicato per ogni coefficiente b).

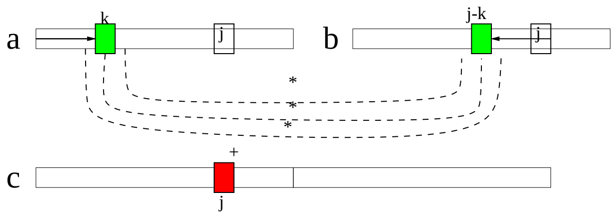


Figura 3.1: La convoluzione di due vettori.

Il vettore di coefficienti c che rappresenta il prodotto di due polinomi è anche detto **convoluzione** dei vettori in ingresso a e b, denotata come

$$c = a \otimes b$$

.

3.2.2 Rappresentazione punto-valore

La rappresentazione punto-valore consiste in n "esempi" della mappa eseguita dalla funzione (coppie ingresso-uscita).

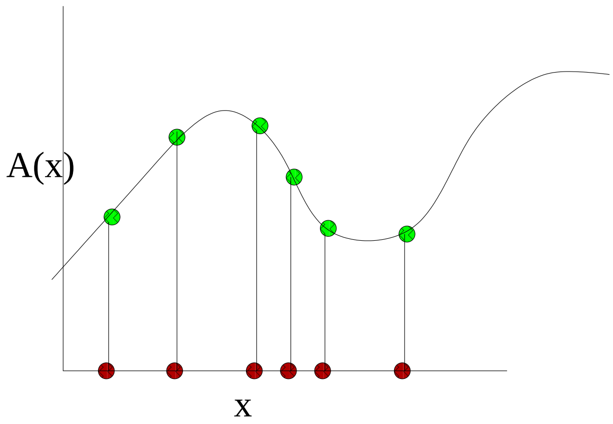


Figura 3.2: La rappresentazione punto-valore dei polinomi.

Precisamente, si tratta di n **paia punto-valore**

$$\{(x_0, y_0), (x_1, y_1), (x_{n-1}, y_{n-1})\}$$

tali che tutti i valori x_k sono distinti e $y_k = A(x_k)$. La rappresentazione non è univoca.

Il calcolo di una rappresentazione punto-valore richiede n valutazioni del polinomio, quindi $\Theta(n^2)$ in totale se si usa la regola di Horner (si vedrà che con una scelta oculata di particolari valori di x_k il calcolo può essere accelerato fino a $\Theta(n \log n)$).

L'operazione inversa della valutazione (ricostruire i coefficienti di un polinomio) da n coppie punto-valore, si chiama **interpolazione**.

Data una rappresentazione punto-valore, il polinomio interpolante è unico. Si tratta di risolvere un sistema di equazioni lineari: n equazioni in n incognite[1].

Il sistema è risolvibile in tempo $O(n^3)$, usando la decomposizione LU della matrice. Una tecnica più veloce è basata sulla formula di Lagrange, che richiede $\Theta(n^2)$ per calcolare tutti i coefficienti.

La rappresentazione punto-valore è comoda per eseguire somme e **moltiplicazioni** di polinomi (purchè polinomi diversi siano rappresentati scegliendo gli stessi valori $x_0, x_1, ..., x_{n-1}$). Per rappresentare il prodotto servono $2n$ coppie punto-valore, è quindi richiesta un'estensione delle rappresentazioni punto-valore di A e di B. Ad esempio, la rappresentazione estesa di A è:

$$\{(x_0, y_0), (x_1, y_1), (x_{2n-1}, y_{2n-1})\}$$

La moltiplicazione di polinomi nella rappresentazione punto-valore richiede *meno* della moltiplicazione nella rappresentazione con coefficienti, si passa da $\Theta(n^2)$ a $\Theta(n)$

3.2.3 Moltiplicazione veloce di polinomi nella rappresentazione con coefficienti

È possibile sfruttare la moltiplicazione "veloce" nella rappresentazione punto-valore per accelerare la moltiplicazione nella rappresentazione con coefficienti?

Sì, purchè si scelgano furbamente i punti di valutazione. Se si usano le **radici complesse dell'unità** è possibile produrre la rappresentazione punto-valore a partire dalla rappresentazione con coefficienti in tempo $\Theta(n \log n)$. Il passaggio si chiama Trasformata Discreta di Fourier (DFT). L'operazione inversa (inverse - DFT) riporta la rappresentazione nella forma con coefficienti.

La **Trasformata Veloce di Fourier** (Fast Fourier Transform – FFT) esegue la trasformazione diretta ed inversa in tempo $\Theta(n \log n)$. La strategia è illustrata in Fig. 3.3.

Per semplicità assumeremo che n sia una potenza di due e che, quando serve, si aggiungano dei coefficienti uguali a zero.

La strategia per la moltiplicazione veloce è la seguente:

[1] Si tratta della generalizzazione del noto principio per il quale per due punti nel piano passa una ed una sola retta – purchè ovviamente i punti siano distinti.

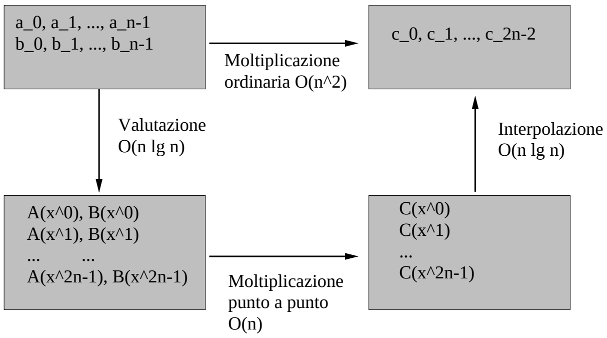


Figura 3.3: Come moltiplicare velocemente i polinomi.

1. Aumenta il grado di $A(x)$ e $B(x)$ portando entrambi a grado $2n - 1$ (aggiungendo coefficienti uguali a zero).
2. Valuta i due polinomi usando le $2n$ radici dell'unità (FFT).
3. Moltiplica punto a punto
4. Crea la rappresentazione con coefficienti di $C(x)$ applicando ancora FFT per eseguire l'inversa della DFT.

È facile dimostrare che il ciclo completo richiede $\Theta(n \log n)$ una volta che si dimostra che la FFT richiede tempo $\Theta(n \log n)$.

3.3 La DFT e la FFT

Vediamo ora come eseguire la valutazione e l'interpolazione in tempo $\Theta(n \log n)$ usando le radici complesse dell'unità.

3.3.1 Radici complesse dell'unità

Una **radice complessa n-esima dell'unità** è un numero complesso ω tale che:

$$\omega^n = 1$$

Ci sono n radici complesse n-esime dell'unità: $e^{\frac{2\pi i k}{n}}$, per $k = 0, 1, ..., n-1$

Le n radici complesse sono rappresentabili in un piano, usando la relazione:

$$e^{i\theta} = \cos(\theta) + i \sin(\theta)$$

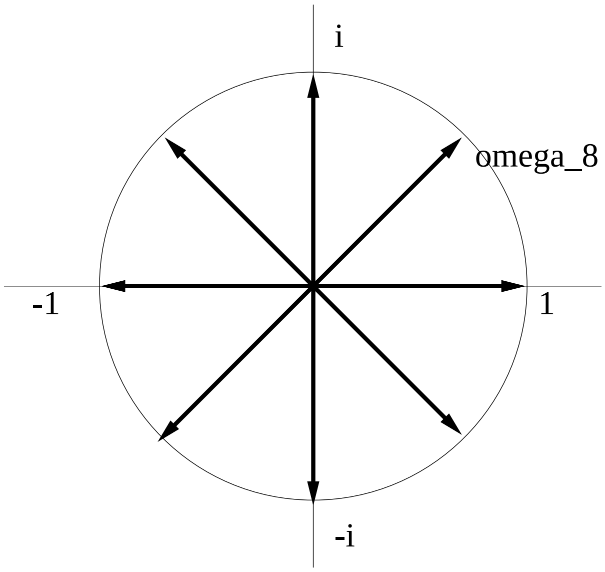


Figura 3.4: Radici complesse dell'unità: il caso $n = 8$.

Il valore: $\omega_n = e^{\frac{2\pi i}{n}}$ è detto **radice principale n-esima dell'unità**. Le altre radici complesse n-esime sono potenze di questa.

Le n radici complesse n-esime dell'unità: $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$ sono un gruppo sotto la moltiplicazione.

È facile controllare le seguenti proprietà:

- **Lemma della cancellazione**: per ogni intero $n \geq 0, k \geq 0$ e $d > 0$

$$\omega_{dn}^{dk} = \omega_n^k$$

- per ogni $n > 0$:

$$\omega_n^{n/2} = \omega_2 = -1$$

- **Lemma del dimezzamento**: se $n > 0$ è pari, i quadrati delle n radici complesse n-esime dell'unità sono le $n/2$ radici $(n/2)$-esime dell'unità.

- **Lemma della somma**: per ogni $n \geq 1$ ed intero non-negativo k non divisibile per n:

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

Si usa l'equazione:

$$\sum_{j=0}^{n-1}(\omega_n^k)^j = \frac{(\omega_n^k)^n - 1}{\omega_n^k - 1}$$

Il denominatore è diverso da zero perchè k non è divisibile per n per le ipotesi fatte [2].

3.3.2 La DFT

Si vuole valutare il polinomio:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

in corrispondenza delle n radici complesse n-esime dell'unità: $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$ Assumiamo n potenza di due, sai A dato nella rappresentazione con coefficienti: $a = (a_0, a_1, ..., a_{n-1})$.

Definiamo come y_k i singoli risultati della valutazione:

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj}$$

Il vettore $y = (y_0, y_1, ..., y_{n-1})$ è la **Trasformata discreta di Fourier** (DFT) del vettore di coefficienti $a = (a_0, a_1, ..., a_{n-1})$. Scriveremo $y = DFT(a)$

3.3.3 La FFT

La **Trasformata Veloce di Fourier (FFT)** è un algoritmo efficiente, con tempo $\Theta(n \log n)$, per il calcolo della DFT che sfrutta le proprietà della radici dell'unità.

La FFT usa una strategia di *divide et impera*, definendo prima due sottoproblemi ottenuti considerando i coefficienti pari e dispari di $A(x)$. I due sottoproblemi sono dati dal calcolo dei due polinomi di grado $n/2-1$:

$$A^{[0]}(x) = a_0 + a_2 x + a_4 x^2 + ... + a_{n-2} x^{n/2-1}$$

[2] Dal punto di vista "visivo": trovo il baricentro di punti distribuiti in modo uniforme attorno all'origine: esso risulta zero.

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + ... + a_{n-1}x^{n/2-1}$$

è immediato controllare che:

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) \tag{3.1}$$

Concludendo, il problema di valutare $A(x)$ sulle n radici complesse n-esime dell'unità: $\omega_n^0, \omega_n^1, ..., \omega_n^{n-1}$ è riducibile all'esecuzione dei dei due passi:

1. valutare i due polinomi di grado $n/2 - 1$: $A^{[0]}(x)$ ed $A^{[1]}(x)$ in corrispondenza dei punti:

$$(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2 \tag{3.2}$$

2. combinare i risultati secondo l'equazione 3.1

Ora, per il lemma del dimezzamento, i valori diversi nella lista 3.2 sono in realtà le $n/2$ radici $n/2$-esime dell'unità.

I sottoproblemi hanno quindi la stessa forma del problema originale, ma la dimensione è dimezzata.: abbiamo decomposto il calcolo di una DFT_n su n elementi al calcolo di due $DFT_{n/2}$ su $n/2$ elementi.

La decomposizione è alla base dell'algoritmo ricorsivo FFT, che calcola la DFT di un vettore ad n elementi $a = (a_0, a_1, ..., a_{n-1})$, dove n è una potenza di due.

```
#include "Complex.H"
#include <iostream.h>
#include <math.h>

double pi = 2.0 * asin(1.0);

int RecursiveFFT(complex *a, int n, complex *result)
{
if(n == 1) {
result[0] = a[0];
return 1;
}

complex omega_n(cos(2*pi / n), sin(2*pi / n));

complex omega(1.0);

complex *a0 =  new complex [n/2];
complex *a1 =  new complex [n/2];
complex *y0 =  new complex [n/2];
complex *y1 =  new complex [n/2];


for(int i=0; i < n/2; i++){
a0[i] = a[2*i];
a1[i] = a[2*i+1];
}

RecursiveFFT(a0, n/2, y0);
RecursiveFFT(a1, n/2, y1);

for(int k=0; k< n/2; k++){
result[k] = y0[k] + omega * y1[k];
result[k + (n/2)] = y0[k] - omega * y1[k];
omega = omega * omega_n;
}

delete [] a0;
delete [] a1;
delete [] y0;
delete [] y1;

return 1;
}
```

La procedura `RecursiveFFT` lavora in questo modo. La base della ricorsione è immediata:

$$y_0 = a_0 \omega_1^0 = a_0$$

Per l'aggiornamento di ω, quando viene eseguito il ciclo iterativo per la k-esima iterazione $\omega = \omega_n^k$.

Il calcolo ricorsivo per $k = 0, 1, ..., n/2 - 1$ avviene usando il lemma della cancellazione:

$$y_k^{[0]} = A^{[0]}(\omega_{n/2}^k) = A^{[0]}(\omega_{n/2}^k) y_k^{[1]} = A^{[1]}(\omega_{n/2}^k) = A^{[1]}(\omega_{n/2}^k)$$

Per comprendere l'ultima linea si noti che, per $k = 0, 1, ..., n/2 - 1$:

$$\begin{aligned}
y_{k+(n/2)} &= y_k^{[0]} - \omega_n^k y_k^{[1]} && (3.3)\\
&= y_k^{[0]} + \omega_n^{k+(n/2)} y_k^{[1]} && (3.4)\\
&= A^{[0]}(\omega_n^{2k}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k}) && (3.5)\\
&= A^{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A^{[1]}(\omega_n^{2k+n}) && (3.6)\\
&= A(\omega_n^{k+(n/2)}) && (3.7)\\
& && (3.8)
\end{aligned}$$

3.4 Realizzazioni efficienti della FFT

Capitolo 4

ALGORITMI INGORDI (GREEDY)

4.1 Struttura dati Heap

La struttura dati chiamata **Heap** è un array A che può essere visto come un albero binario completo, che per ogni nodo tranne che per la radice soddisfa la **proprietà di Heap**:

A[Parent(i)] $>=$ $A[i]$

La heap realizza una coda con priorità.

Dell'array A ci interessa conoscere il numero dei suoi elementi, che chiamiamo *lung*[A], ed *heap-size*[A] che è il numero degli elementi della struttura heap memorizzata in A. Come si vede nella figura 4.1, $A[1]$ è la radice dell'albero. Dato poi l'i-esimo nodo dell'albero, si ha che:

Parent(i) = $A[i/2]$

Left(i) = $A[2i]$

Right(i) = $A[2i+1]$

Sempre facendo riferimento all'esempio illustrato in fig. 4.1 avremo *heap-size*[A] = 12 e *lung*[A] = 14.

La funzione fondamentale per operare con una struttura heap è la cosiddetta funzione **Heapify**. Essa riceve in input l'array A e un indice i. Quando è chiamata, Heapify suppone che i sottoalberi che hanno radice in Left(i) e Right(i) godano della proprietà di heap, ma il nodo $A[i]$ potrebbe essere più piccolo dei suoi figli. Heapify "spinge verso il basso" $A[i]$ e ripristina la proprietà di heap su tutto l'albero.

Lo pseudo codice per Heapify è:

```
HEAPIFY(A, i)
  l ← Left(i);
  r ← Right(i);
  if l <= heap-size and A[l] > A[i];
```

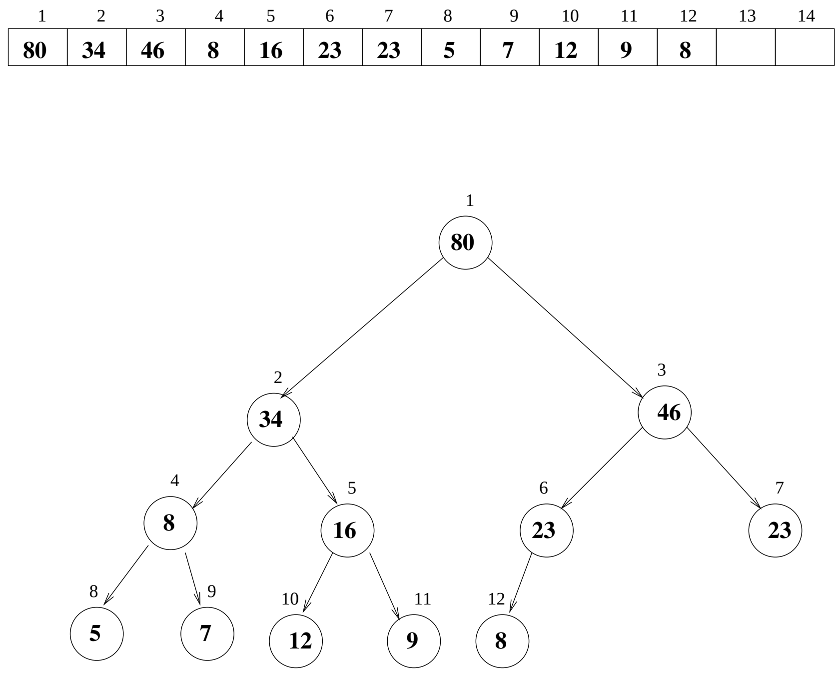


Figura 4.1: Heap visto come array e come albero.

```
    then largest ← l;
    else largest ← i;
if r <= heap-size and A[r] > A[largest];
    then largest ← r;
if largest != i;
    then exchange A[i] ← A[largest];
    HEAPIFY(A, largest);
```

La fig. 4.2 illustra il funzionamento della funzione Heapify chiamata per $i = 2$.

4.2 Codice di Huffman

Il **codice di Huffman** è una tecnica di compressione dati molto usata ed efficiente. Sfrutta una tabella della frequenza delle occorrenze di ogni carattere per arrivare ad una rappresentazione ottimale dei caratteri come stringa binaria.

Partiamo con un esempio: supponiamo di voler comprimere una stringa di 80 caratteri, formata solo con le lettere *a, b, c, d, e, f, g* , senza spazi bianchi.

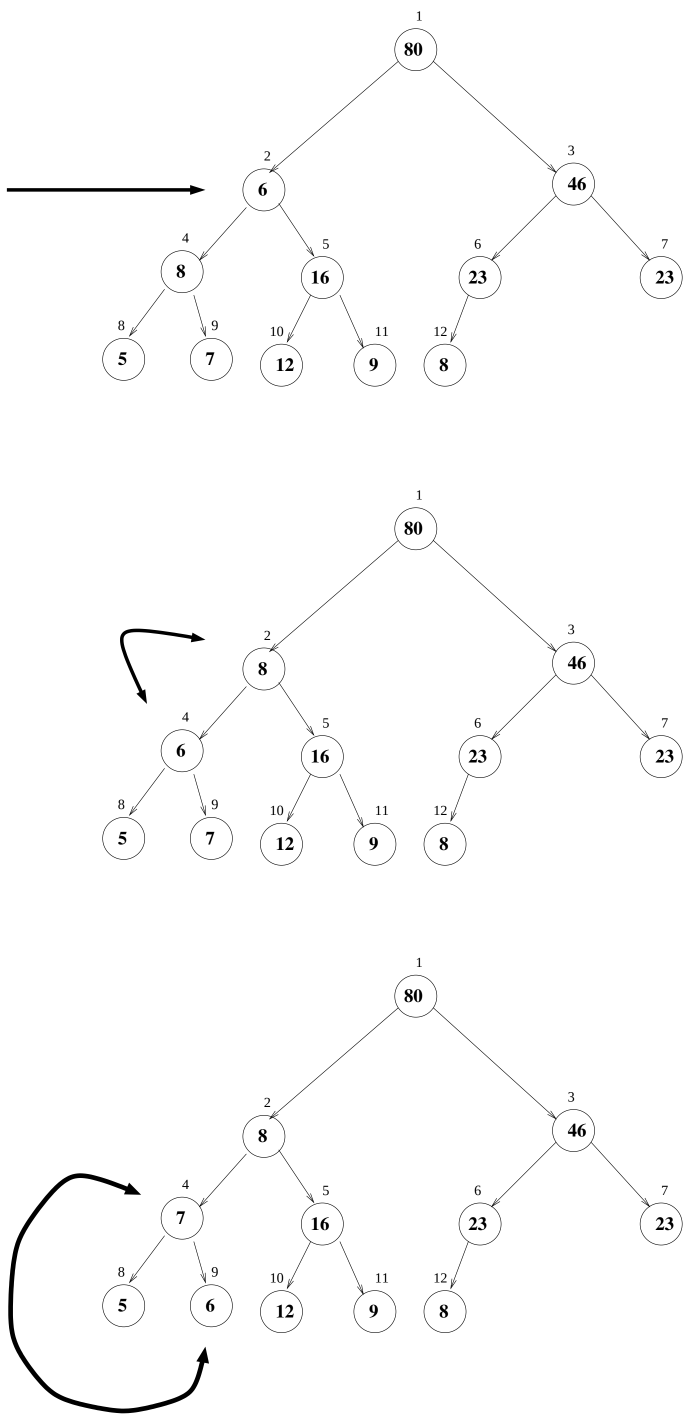


Figura 4.2: Funzionamento di Heapify.

Un primo modo per comprimere i dati consiste nello sfruttare il fatto che nel nostro esempio abbiamo solo 7 lettere. Possiamo rappresentarle usando solo tre bit: a = 000, b = 001, c = 010, d = 011, e = 100, f = 101, g = 111. In questo modo la stringa di 80 caratteri diventa una stringa di 240 bit (piú i separatori fra ogni terna di bit).

Si può fare di meglio, se sfruttiamo il fatto che alcuni caratteri compaiono piú frequentemente di altri. Come primo passo quindi si deve costruire una tabella di frequenza. Supponiamo di ottenere la tabella seguente:

lettera:	a	b	c	d	e	f	g
frequenza:	23	11	12	8	18	5	3

Tabella 4.1: Tabella di frequenza

Usiamo una rappresentazione a lunghezza variabile e rappresentiamo i caratteri piú frequenti con meno bit. Per esempio: a = 0, b = 10, c = 01, d = 000, e = 00, f = 11, g = 001. In questo caso abbiamo bisogno di 148 bit, ai quali però vanno sempre sommati dei separatori tra ogni gruppo di bit, per evitare ambiguità nella decodifica (il codice 01000 rappresenta *abe* oppure *cd* ?). Si può ulteriormente migliorare il risultato eliminando la necessità di separare (o comunque distinguere in qualche modo) ogni gruppo di bit. Lo si fa usando un **codice prefisso**, nel quale ogni codice rappresentante una lettera non è la prima parte (il prefisso) del codice di un'altra lettera. Questo permette di scrivere una unica stringa di bit senza interruzioni, con possibilità di decodificarla senza ambiguità. Ad esempio, nel nostro caso: a = 0, b = 101, c = 100, d = 1101, e = 111, f = 11000, g = 11001. Risultato: 218 bit.

Supponiamo di avere un insieme C di caratteri con relative frequenze. La struttura ideale per allocarli in vista della compressione con l'algoritmo di Huffman è una coda con priorità, che chiamiamo Q, i cui elementi sono nodi di albero binario ordinati in modo crescente. In ogni nodo viene memorizzato un carattere e la relativa frequenza. L'algoritmo costruisce un albero con strategia bottom-up.

Lo pseudo codice è:

HUFFMAN
  $n \leftarrow dim(C)$;
  $Q \leftarrow C$;
  for $i \leftarrow 1$ **to** n-1;
    $z \leftarrow CreaNodo()$;
    $x \leftarrow left[z] \leftarrow Min(Q)$;
    $y \leftarrow right[z] \leftarrow Min(Q)$;
    $Freq(z) \leftarrow Freq(x) + Freq(y)$;

```
    └ Insert(Q,z);
└ return Min(Q);
```

Nelle figure 4.3, 4.4 e 4.5 viene illustrato come lavora l'algoritmo. Ad ogni passo i due nodi con frequenza minore, per esempio nel primo passaggio g ed f, diventano i figli di un nuovo nodo che ha come valore la somma delle frequenze dei figli e la lista viene riordinata. In presenza di più di due nodi con frequenza minima, la scelta è arbitraria ed ininfluente sul risultato. Alla fine si rimane con un unico albero. Assegnando 0 ai rami sinistri ed 1 ai destri, percorrendo l'albero fino ad ogni carattere otteniamo la sua codifica, che si può dimostrare ottimale.

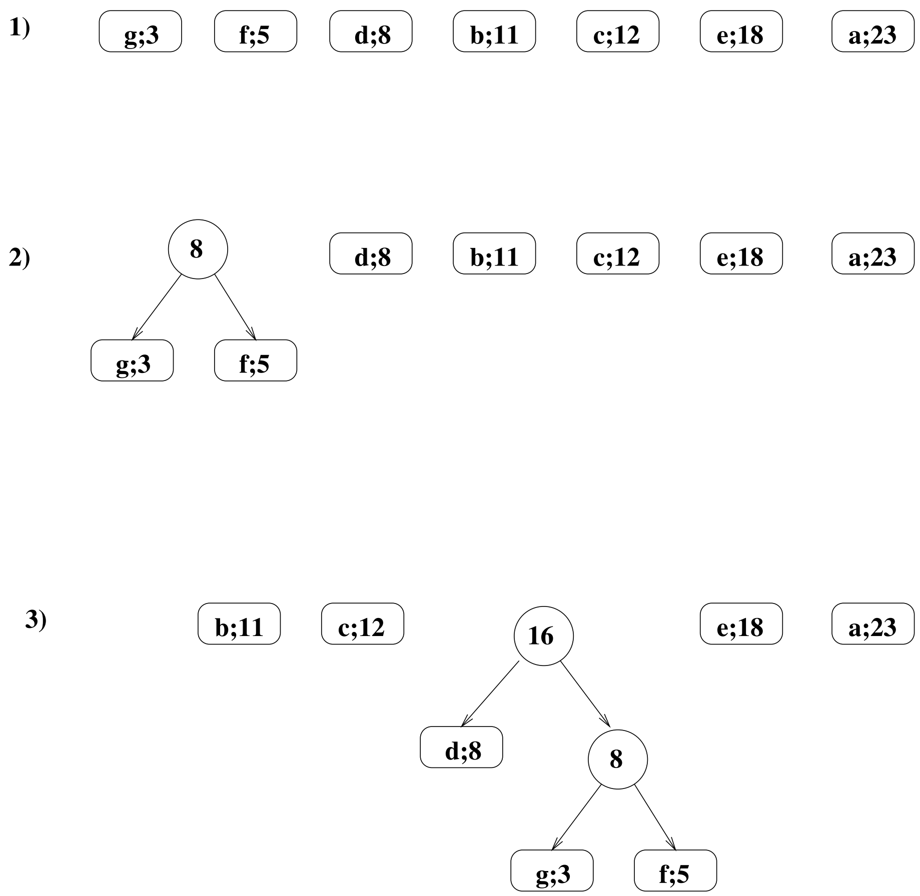


Figura 4.3: Primi 3 passi dell'algoritmo di Huffman.

Nel nostro esempio otteniamo che $a = 10$, $b = 110$, $c = 111$, $d = 010$, $e = 00$, $f = 0111$, $g = 0110$. La stringa (*begafa*) viene quindi codificata come (11000100110100111110).

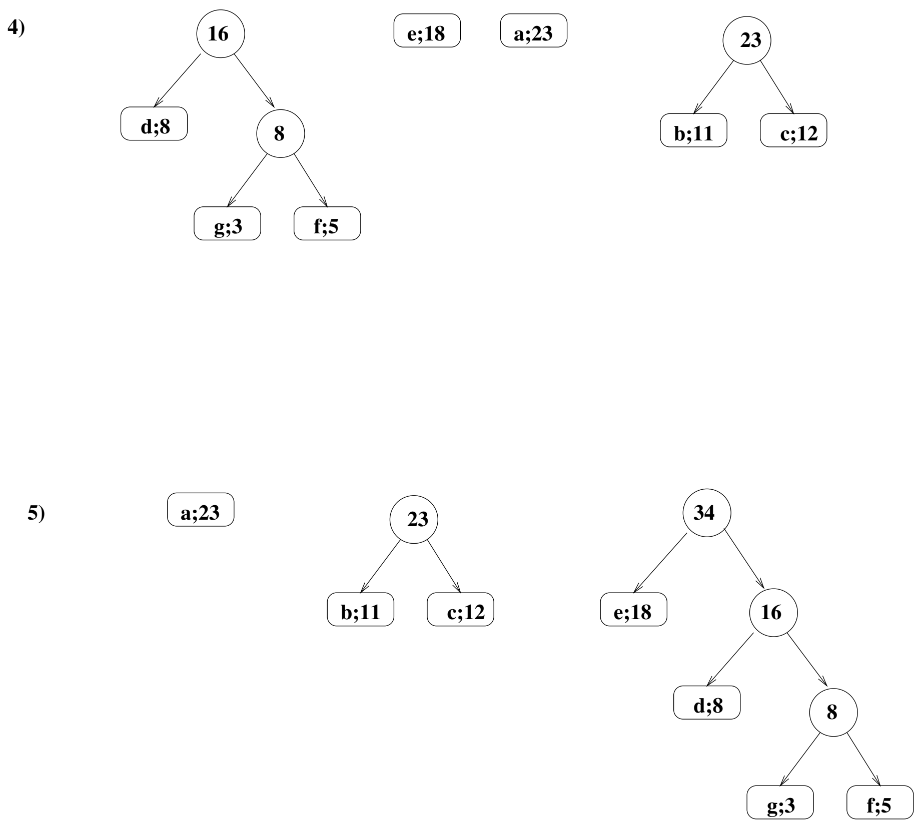


Figura 4.4: Passi 4 e 5 dell'algoritmo di Huffman.

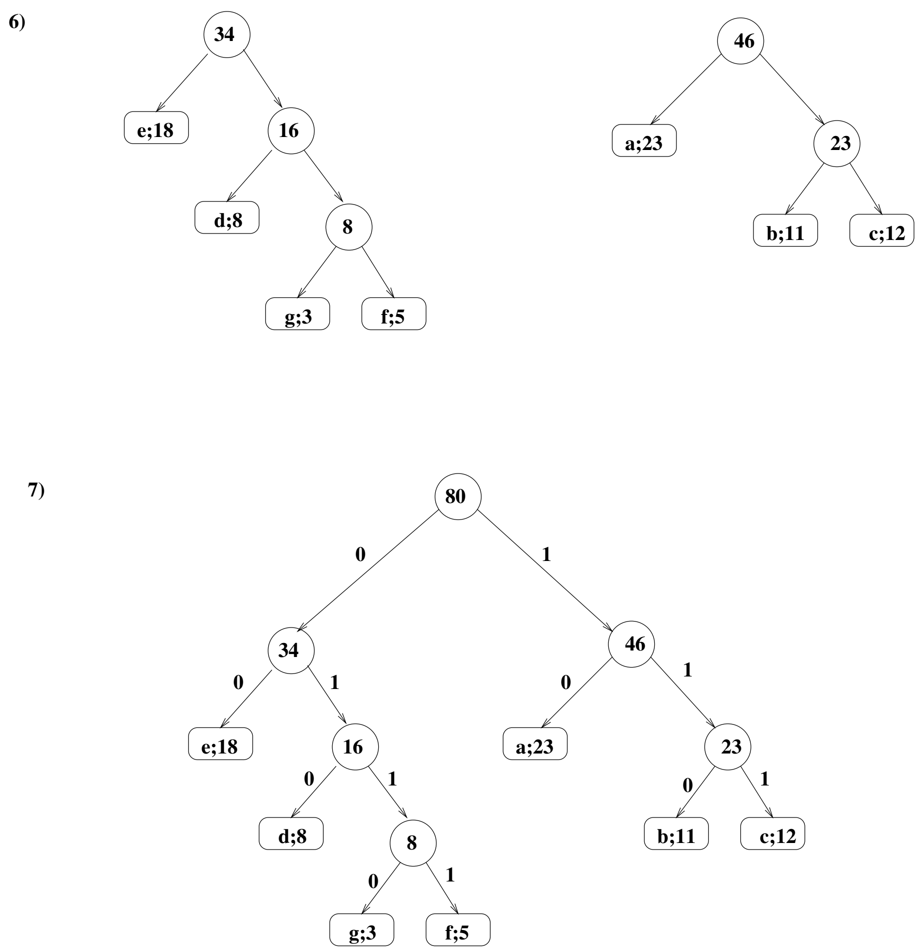


Figura 4.5: Passi 6 e 7 dell'algoritmo di Huffman.

Capitolo 5

PROGRAMMAZIONE DINAMICA

Le tecniche di programmazione dinamica si applicano a certi problemi di ottimizzazione, cioè problemi che richiedono di massimizzare o minimizzare certe quantità, che si possono spezzare in sottoproblemi. Puó capitare che lo stesso sottoproblema compaia più volte nella scomposizione del problema principale. A differenza delle strategie di tipo *Divide et Impera*, che risolverebbero piú volte lo stesso sottoproblema, l'idea in un algoritmo di programmazione dinamica è di risolvere prima i sottoproblemi base (strategia bottom-up) mantenendo in memoria i risultati parziali ottenuti, e quindi passare alla risoluzione a livelli sempre più alti utilizzando i risultati già in memoria. La tecnica della programmazione dinamica è stata introdotta nel 1957 da Bellman.

5.1 Prodotto in cascata di matrici

Si supponga di voler moltiplicare fra loro una serie di matrici. Il numero di prodotti scalari da effettuare varia, e puó variare anche di molto, a seconda dell'ordine in cui si effettuano le moltiplicazioni. Per esempio, se per moltiplicare una matrice A 4x2 con B 2x3 e con C 3x1 si fa il prodotto ((AxB)xC), si calcolano 24+12 prodotti scalari. Se invece si sceglie di procedere da destra verso sinistra, cioé facendo (Ax(BxC)), i prodotti saranno 6+16. Vale quindi la pena di perdere un po' di tempo alla ricerca di una buona sequenza di moltiplicazione.

Si supponga di voler moltiplicare N matrici $M_1 M_2 M_3 ... M_N$, dove la matrice M_i abbia r_i righe e r_{i+1} colonne. Si cercherà la sequenza di moltiplicazione (la parentesizzazione completa) che minimizza il numero dei prodotti scalari.

Per comodità si scriva $M_{i..j}$ per indicare la matrice che risulta dal prodotto $M_i M_{i+1}$ ... M_j. Si utilizzerà una matrice m: in m[i,j] si memorizzerà il minimo numero di moltiplicazioni necessarie per calcolare $M_{i..j}$. I valori di m sulla diagonale saranno tutti uguali a 0 ($M_{i..i} = M_i$) e si userà solo la parte sopra alla diagonale. Nell'array r si memorizzino le dimensioni delle matrici da moltiplicare.

Si inizia calcolando il costo (= numero di moltiplicazioni) per la moltiplicazione delle coppie di matrici $M_{1..2} = M_1M_2$, $M_{3..4} = M_3M_4$, $M_{5..6} = M_5M_6$, ..., $M_{N-1..N} = M_{N-1}M_N$. Poi si passa a cercare il modo migliore per calcolare le triple successive di matrici, utilizzando i risultati già trovati per le coppie. Per esempio, le parentesizzazioni possibili per la tripla $M_1M_2M_3$ sono due: $((M_1M_2)M_3)$ oppure $((M_1M_2)M_3)$. Il costo di (M_1M_2) è già stato calcolato e si trova in m[1,2]. Quindi si ha:

costo di $((M_1M_2)M_3)$ = m[1,2] + r[1]*r[2]*r[3]

In seguito si prendono in considerazione i gruppi successivi di quattro matrici, e cosí via. Nella matrice m, in m[1,2] si avrà il numero di moltiplicazioni minimo per calcolare M_1M_2, in m[1,3] il numero di moltiplicazioni minimo per calcolare $M_1M_2M_3$, in m[1,N] il numero di moltiplicazioni minimo per calcolare $M_1M_2...M_N$.

Lo pseudo codice è:

```
1.  MATRIX-CHAIN (M_1..N);
2.  for i ← 1 to N;
3.  |  do for j ← i+1 to N;
4.  |  |  do m[i,j] ← ∞;
5.  for i ← 1 to N;
6.  |  do m[i,j] ← 0;
7.  for j ← 1 to N;
8.  |  do for i ← 1 to N-j;
9.  |  |  do for k ← i+1 to i+j;
10. |  |  |  do tmp ← m[i,k-1] + m[k,i+j] + r_i r_k r_{i+j+1};
11. |  |  |  |  if tmp < m[i,i+j];
12. |  |  |  |  |  then m[i,i+j] ← tmp;
13. |  |  |  |  |  s[i,i+j] ← k;
```

Le righe 3 e 4 nello pseudocodice semplicemente inizializzano la matrice m al massimo numero ammissibile. Il successivo ciclo for pone a zero tutti gli elementi sulla diagonale di m.

Notare l'uso di una matrice di supporto s: quando si è trovata la migliore suddivisione in due del prodotto $M_{i..i+j}$, cioè un numero k tale che la moltiplicazione di $M_{i..k-1}M_{k..i+j+1}$ sia la migliore per k che varia tra i e i+j, si memorizza k in s[i,i+j]. Assieme alla matrice m, questa serve per ricostruire la migliore parentesizzazione del prodotto che si vuole calcolare. Lo si fa con una procedura ricorsiva come segue:

```
MATRIX-CHAIN-ORDER (i, j, s);
```

```
if i = j
  then out ← M(i);
  else
    out ← "(";
    MATRIX-CHAIN-ORDER (i, s[i][j]-1, s);
    MATRIX-CHAIN-ORDER (s[i][j], j, s);
    out ← ")";
```

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2		0	2625	4375	7125	10500
3			0	750	2500	5375
4				0	1000	3500
5					0	5000
6						0

m

	1	2	3	4	5	6
1		1	1	3	3	3
2			2	3	3	3
3				3	3	3
4					4	5
5						5
6						

s

Figura 5.1: Moltiplicazione di matrici: come risultano le matrici di supporto m ed s dopo l'applicazione dell'algoritmo sulle seguenti matrici: $A_1 = 30\text{x}35, A_2 = 35\text{x}15, A_3 = 15\text{x}5, A_4 = 5\text{x}10, A_5 = 10\text{x}20, A_6 = 20\text{x}25$.

5.2 String matching approssimato

Un'altro problema a cui può essere applicato un algoritmo di programmazione dinamica è il cosiddetto **String Matching Approssimato**.

Siano P e T due sequenze di m ed n caratteri rispettivamente, con $m <= n$. Il problema consiste nel trovare una occorrenza di P in T ammettendo anche un numero limitato di differenze. Le differenze ammesse saranno:

Sostituzione: un carattere è diverso.

Inserimento: in P c'è un carattere che non compare in T.

Cancellazione: un carattere in T non compare in P.

In genere, per comodità di implementazione dell'algoritmo, si suppone che che sia P che T abbiano un primo carattere vuoto. Come al solito, si usa una matrice D in cui memorizzare i risultati parziali. Di seguito lo pseudocodice:

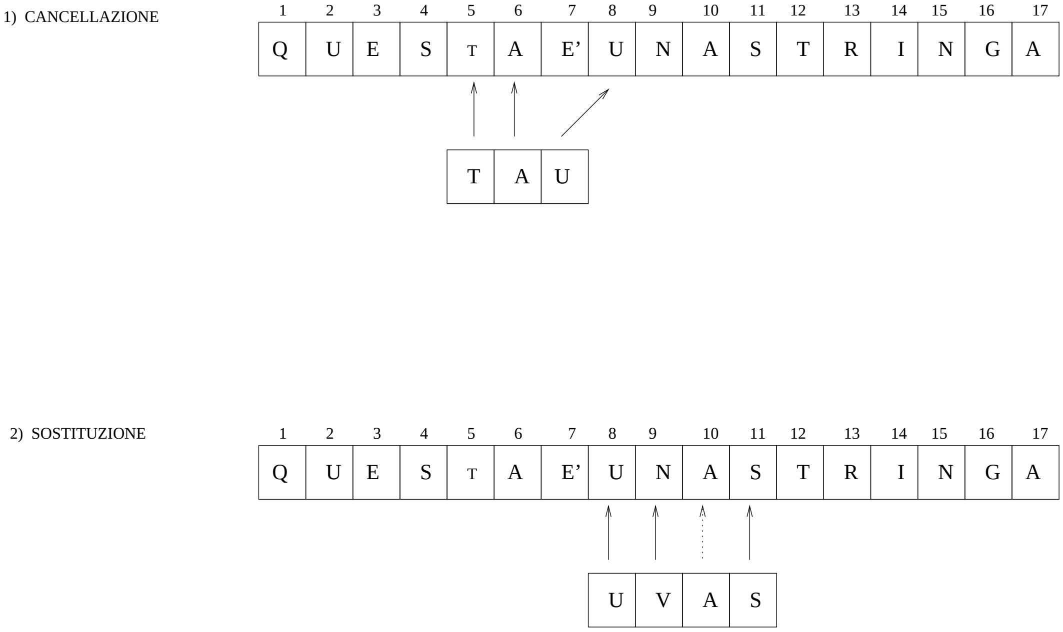


Figura 5.2: String matching approssimato: in 1) la stringa "TAU" ha una occorrenza a partire dalla posizione 5 di T con un differenza di tipo cancellazione. In 2) la stringa "UVAS" ha una occorrenza dalla posizione 8 con una sostituzione. In entrambi i casi si parla di **occorrenza 1-approssimata**.

```
STRING-MATCHING-APPROX (P, T);
for i ← 0 to n
  D[0,i] ← 0;
for i ← 0 to m
  D[i,0] ← i;
for i ← 1 to m
  for j ← 1 to n
    if P[i] = T[j]
      min ← D[i-1, j-1];
    else
      min ← D[i, j-1] + 1;
    if D[i-1, j]+1 < min
      min ← D[i-1, j]+1;
    if D[i, j-1]+1 < min
      min ← D[i, j-1]+1;
min ← D[m, 0];
for j ← 1 to n
  if D[m, j] < min
```

```
        min ← D[m, j];
        i ← j;
return (i);
```

Capitolo 6

OPERAZIONI I/O SU FILE

Il C++ supporta il sistema di Input/Output del C, ma dispone anche di un proprio sistema ad oggetti molto complesso e potente, del quale nel seguito si vedranno solo alcune delle caratteristiche base. Per approfondire e sfruttare al meglio le possibilità offerte in questo campo dal C++ fare riferimento a qualunque manuale del linguaggio.

Per leggere o scrivere su file, in C++ bisogna aprire uno **stream** (= flusso) ed associarlo al file. Uno stream è un dispositivo logico che fa da interfaccia al dispositivo fisico a cui è associato. Gli stream sono largamente indipendenti dal dispositivo fisico, per cui le stesse funzioni di I/O possono essere utilizzate per dispositivi fisici diversi: per esempio per scrivere su disco, nastro, schermo, stampante. Esistono tre tipi di stream: di input, di output e di input/output.

All'inizio dell'esecuzione di un programma in C++ vengono aperti automaticamente alcuni stream predefiniti. I più importanti sono *cin* e *cout*, collegati rispettivamente alla tastiera e allo schermo.

Per creare ed eseguire operazioni su uno stream si deve includere nel programma l'intestazione *fstream.h*:

```
#include <fstream.h>
```

Fstream.h contiene la definizione di numerose classi, tra le quali le seguenti sono da usare per creare un oggetto stream:

ifstream $\longmapsto$ operazioni di input

ofstream $\longmapsto$ operazioni di output

fstream $\longmapsto$ operazioni di input e output

Il costruttore della classe ofstream crea lo stream di output ed apre il file indicato (o lo crea se non esiste) in modalità scrittura cancellandone il contenuto preesistente.

Ifstream apre il file (o lo crea se non esiste) in modalità lettura, posizionandosi all'inizio.

Fstream apre il file (o lo crea se non esiste) in modalità lettura e scrittura, posizionandosi all'inizio.

Generalmente è opportuno controllare immediatamente se l'operazione di creazione stream e apertura file è andata a buon fine.

Esempio: creazione dello stream di input *miostream* per leggere il contenuto del file *miofile*:

```
ifstream miostream("miofile");
if (!miostream)

     {
     cout << "Impossibile aprire il file !" << endl;
     return 1;
     }

. . .
```

Le operazioni di lettura e scrittura su un file di testo si possono eseguire molto semplicemente con gli operatori << ed >>. Il seguente spezzone di programma apre il file *miofile* in scrittura, vi inserisce tre righe di caratteri prelevati da tastiera e lo chiude. Quindi lo riapre in sola lettura e legge riga per riga il contenuto fino alla fine del file, mandandolo in uscita sullo schermo. Notare l'uso di eof(), una funzione membro di ifstream, ofstream ed fstream.

```
ofstream miostream("miofile");
if (!miostream)

     {
     cout << "Impossibile aprire il file !";
     return 1;
     }

for(i=1; i<= 3; i++)

     {
     cin >> temp;
     miostream << temp << endl;
     }

miostream.close();

ifstream miostream("miofile");
if(!miostream)

     {
     cout << "Impossibile aprire il file !";
```

```
        return 1;
        }

    while(!miostream.eof())

        {
        miostream >> temp;
        cout << temp << endl;
        }

    miostream.close();
    . . .
```

Nella lettura di un file di testo con l'operatore >> avvengono alcune traduzioni di caratteri. Per evitare ciò bisogna aprire il file in modalità binaria ed utilizzare le opportune funzioni, per esempio get() e put():

istream &get(char &*carattere*)

ostream &put(char *carattere*)

La funzione get() legge un singolo carattere dallo stream chiamante e ne inserisce il valore in *carattere*, restituendo l'indirizzo dello stream. La funzione put() scrive *carattere* sullo stream e restituisce l'indirizzo dello stream.

Ecco un esempio in cui viene aperto un file in modalità binaria e con l'uso della funzione get() ne viene letto il contenuto:

```
#include <iostream.h>
#include <fstream.h>

int main( int argc, char *argv[ ] )

    {
    char ch;
    ifstream pippo(argv[1], ios::in | ios::binary);
    if (!pippo)
        {
        cout << "File non aperto";
        return 1;
        }
    while(pippo)
        {
        pippo.get(ch);
        if(pippo)
            cout << ch;
```

```
        }
    return 0;
    }
```

Nota: pippo è uguale a zero quando viene raggiunta la fine del file.

CONTINUA ...