Tecniche Algoritmiche/2
Algoritmi *greedy*

Gianluigi Zavattaro
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna
gianluigi.zavattaro@unibo.it

Introduzione

- Quando applicare la tecnica greedy?
  - Quando è possibile *dimostrare* che esiste una *scelta ingorda*
    - Fra le molte scelte possibili, se ne può facilmente individuare una che porta sicuramente alla soluzione ottima
  - Quando il problema ha sottostruttura ottima
    - “Fatta tale scelta, resta un sottoproblema con la stessa struttura del problema principale”
- Non tutti i problemi hanno una scelta ingorda
  - Quindi non tutti i problemi si possono risolvere con una tecnica greedy

Problema del resto

Problema del resto

- Input
  - Un numero intero positivo *R* che rappresenta un importo (in centesimi di euro) da erogare
- Output
  - Il minimo numero (intero) di monete necessarie per erogare il resto di *R* centesimi di euro, usando solo monete da 50c, 20c, 10c, 5c, 2c e 1c
  - Disponiamo di un numero infinito di monete di ciascun taglio
- Esempio:
  - *R* = 78, 5 pezzi: 50+20+5+2+1
  - *R* = 19, 4 pezzi: 10+5+2+2

Algoritmo greedy per il resto

```
// R = resto da erogare
// T[1..n] = gli n tagli di monete a disposizione
// output = numero totale di monete da erogare
RestoGreedy(integer R, integer T[1..n]) → integer
    ordina-decrescente(T); // ordina i tagli in senso decrescente
    integer nm ← 0;         // numero monete da erogare
    integer i ← 1;
    while ( R > 0 and i ≤ n ) do
        if ( R ≥ T[i] ) then
            R ← R - T[i];
            nm ← nm + 1;
        else
            i ← i + 1;
        endif
    endwhile
    if (R > 0) then
        errore: resto non erogabile
    else
        return nm;
    endif
```

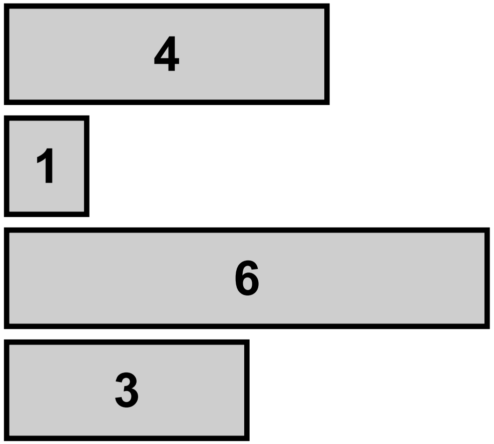
Se il taglio più piccolo disponibile è maggiore di 1c, allora potrebbe non esistere sempre un modo per erogare il resto R (esempio: R=13 usando i tagli T=[10, 5, 2])

Osservazione

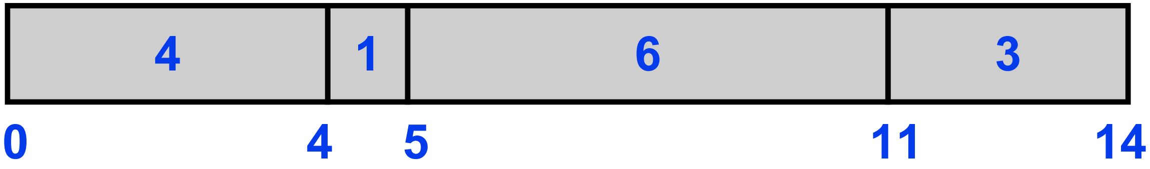
- I sistemi monetari per i quali l'algoritmo greedy fornisce la soluzione ottima si chiamano *sistemi monetari canonici*
  - Xuan Cai (2009). "*Canonical Coin Systems for CHANGE-MAKING Problems*". Proc. Ninth Int. Conf. on Hybrid Intelligent Systems 1: 499–504. doi:10.1109/HIS.2009.103.
- L'algoritmo greedy può fallire con sistemi non canonici
  - Es: erogare 6 con tagli 4, 3, 1 (greedy: 4+1+1, ottimo: 3+3)
  - Es: erogare 6 con tagli 5, 2 (sceglie 5 e poi non può erogare 1, la soluzione 2+2+2 risolverebbe il problema)
- Vedremo più avanti un diverso approccio, basato sulla programmazione dinamica, in grado di determinare sempre la soluzione ottima.

Problema di scheduling (Shortest Job First)

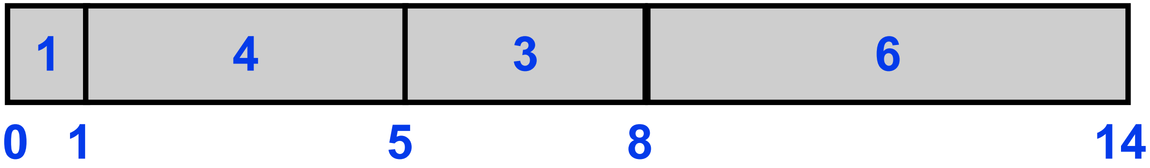
Algoritmo di scheduling—Shortest Job First



- Definizione:
  - 1 processore, n job p_1, p_2, ..., p_n
  - Ogni job p_i ha un tempo di esecuzione $t[i]$
  - Minimizzare il tempo medio di completamento



(4+5+11+14)/4 = 34/4



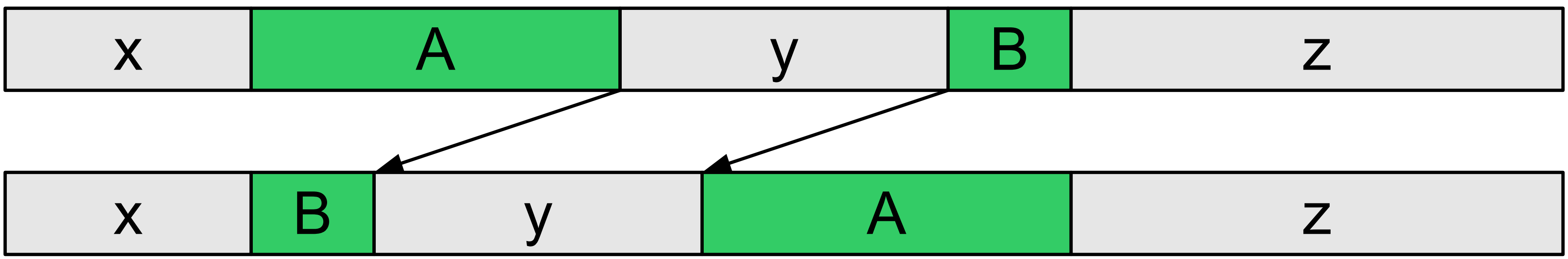
(1+5+8+14)/4 = 28/4

Algoritmo greedy di scheduling

- Siano p_1, p_2, … p_n gli n job che devono essere eseguiti
- L'algoritmo greedy esegue n passi
  - ad ogni passo sceglie e manda in esecuzione il job, tra quelli che rimangono, con il minimo tempo di completamento

Dimostrazione di ottimalità

- Consideriamo un ordinamento dei job in cui un job "lungo" A viene schedulato prima di uno "corto" B
  - x, y e z sono sequenze di altri job



- Osserviamo:
  - Il tempo di completamento dei job in x e in z non cambia
  - Il tempo di completamento di A nella seconda soluzione è uguale al tempo di completamento di B nella prima soluzione
  - Il tempo di completamento di B nella seconda soluzione <u>è minore</u> del tempo di completamento di A nella prima soluzione
  - Il tempo di completamento dei job in y <u>si riduce</u>

Problema della compressione (codifica di Huffman)

Problema della compressione

- Dobbiamo rappresentare sequenze di caratteri (eg. file di testo) secondo una tecnica detta *codifica di caratteri*
  - Si usa una *funzione di codifica f: f(c) = x*
    - *c* è un carattere preso da un alfabeto Σ
    - *x* è una rappresentazione binaria del carattere c
    - *"c* è rappresentato da *x"* in modo efficiente
  - Una sequenza di caratteri $c_1 c_2 .. c_n$ viene codificata con la sequenza di bit $f(c_1)f(c_2)..f(c_n)$
  - Data una qualsiasi codifica, deve essere sempre possibile decodificarla durante la lettura sequenziale *bit-dopo-bit*
- Problema:
  - data la sequenza $c_1 c_2 .. c_n$ definire una funzione di codifica *f* che minimizza la lunghezza della codifica $f(c_1)f(c_2)..f(c_n)$

Codici a lunghezza fissa

- Supponiamo di avere un file di *n* caratteri
  - Possibili car.: **'a' 'b' 'c' 'd' 'e' 'f'**
  - frequenze: **45% 13% 12% 16% 9% 5%**
- Codifica tramite ASCII (8 bit per carattere)
  - Dimensione totale: 8*n* bit
- Codifica basata sull'alfabeto (3 bit per carattere)
  - Codifica: **000 001 010 011 100 101**
  - Dimensione totale: 3*n* bit
- Possiamo fare di meglio?

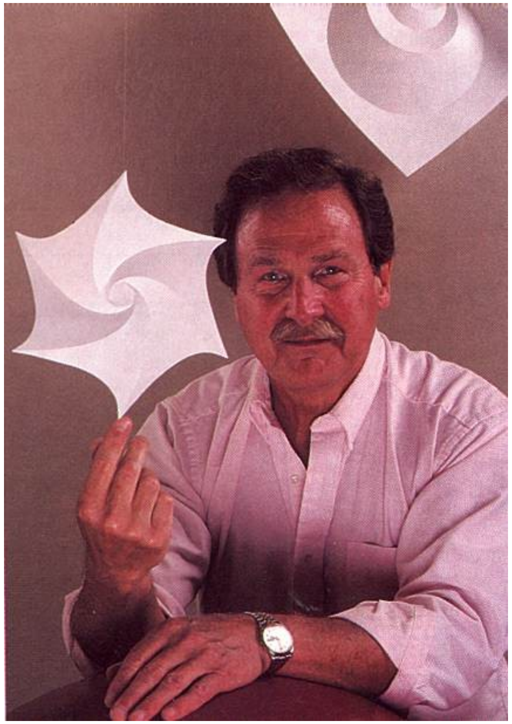
Codici a lunghezza variabile

- Codifica a lunghezza variabile
  - Caratteri: **'a' 'b' 'c' 'd' 'e' 'f'**
  - Codifica: **0 101 100 111 1101 1100**
  - Costo totale: (0.45*1+0.13*3+0.12*3+0.16*3+0.09*4+0.05*4)*n=2.24n
- Codice “a prefisso” (“senza prefissi”):
  - Nessun codice è un prefisso di un altro codice
  - Condizione richiesta per permettere sempre la decodifica durante la lettura bit-dopo-bit
- Esempio: addaabca
  - 0·111·111·0·0·101·100·0

f(a) = 1, f(b)=10, f(c)=101
Come decodificare 101? come “c” o “ba”?

Codici di Huffman

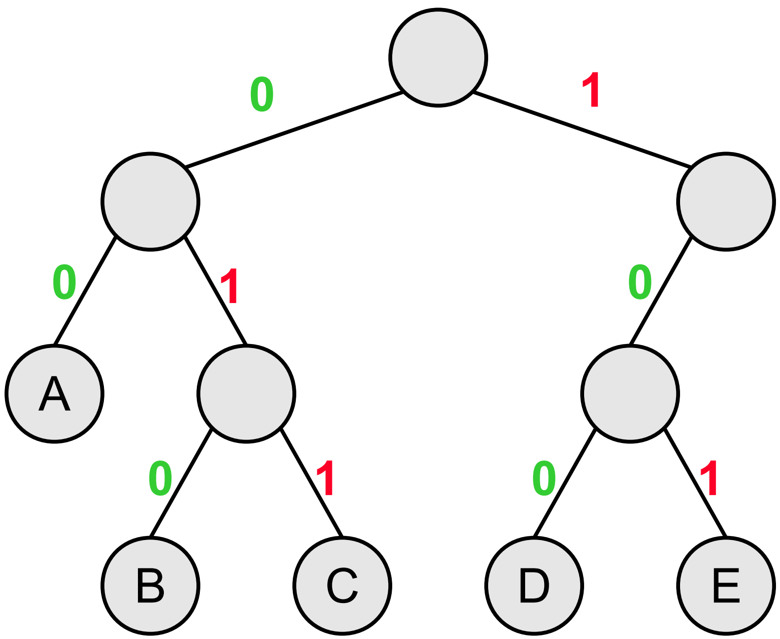
- I codici di Huffman risolvono il problema della compressione
- Huffman, D.A., "*A Method for the Construction of Minimum-Redundancy Codes*," Proc. of the IRE, vol. 40, no. 9, pp. 1098—1101, Sept. 1952, doi: 10.1109/JRPROC.1952.273898



David Albert Huffman
(9 agosto 1925 – 7 ottobre 1999)

Rappresentazione ad albero

- Rappresentazione del codice come un albero binario
  - Figlio sinistro: 0 Figlio destro: 1
  - Caratteri dell'alfabeto sulle foglie



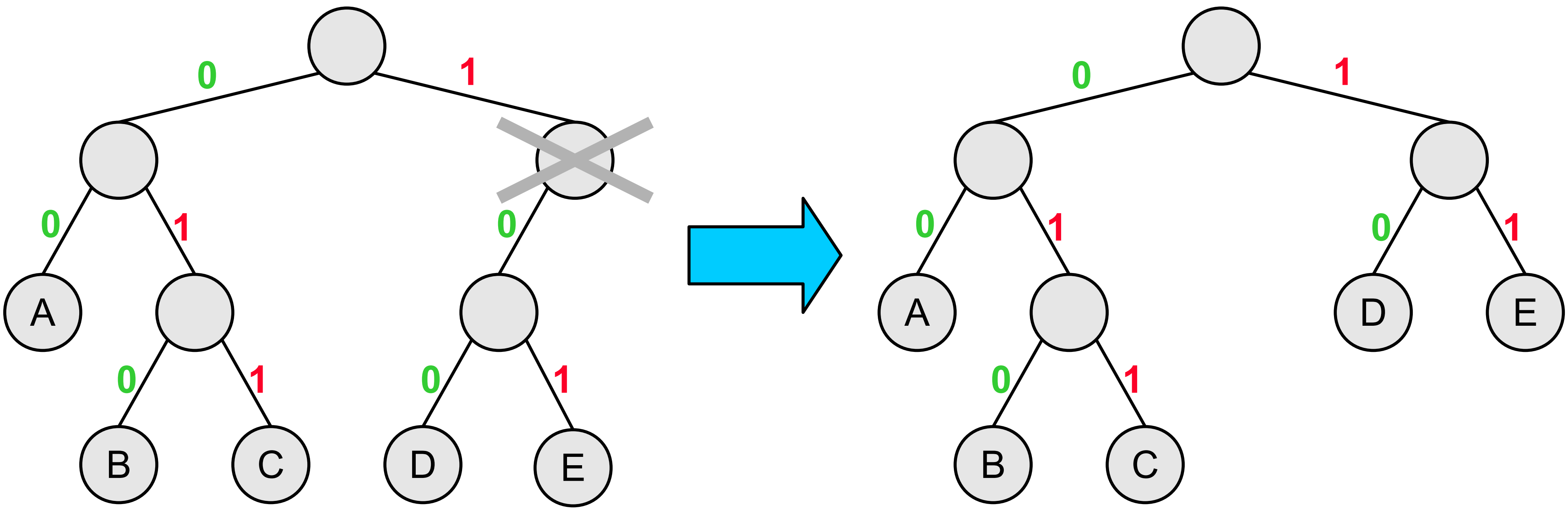
A: 00
B: 010
C: 011
D: 100
E: 101

Algoritmo di decodifica:

1. parti dalla radice
2. leggi un bit alla volta percorrendo l'albero:
   0: sinistra
   1: destra
3. stampa il carattere della foglia
4. torna a 1

Rappresentazione ad albero per la decodifica

- Non c'è motivo per avere un nodo interno con un solo figlio



A: 00
B: 010
C: 011
D: 100
E: 101

A: 00
B: 010
C: 011
D: 10
E: 11

Algoritmo di Huffman

- Principio del codice di Huffman
  - Minimizzare la lunghezza dei caratteri che compaiono più frequentemente
  - Assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero
- Un codice è progettato per un file specifico
  - Si ottiene la frequenza di tutti i caratteri
  - Si costruisce il codice
  - Si rappresenta il file tramite il codice
  - (Si aggiunge al file una rappresentazione del codice per permettere la decodifica)

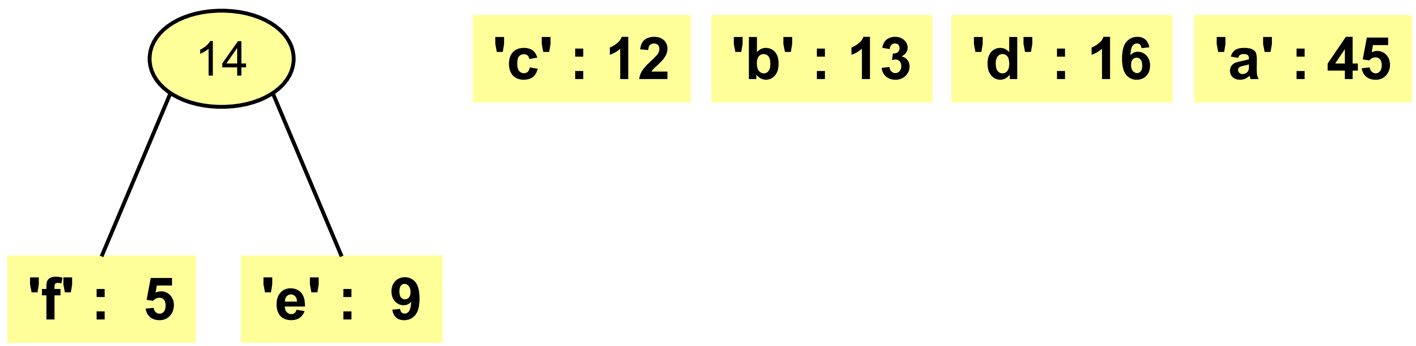
Costruzione del codice

- Passo 1: Costruire una lista ordinata di nodi, in cui ogni nodo contiene un carattere e il numero di volte in cui quel carattere compare nel file

'f' : 5	**'e' : 9**	**'c' : 12**	**'b' : 13**	**'d' : 16**	**'a' : 45**

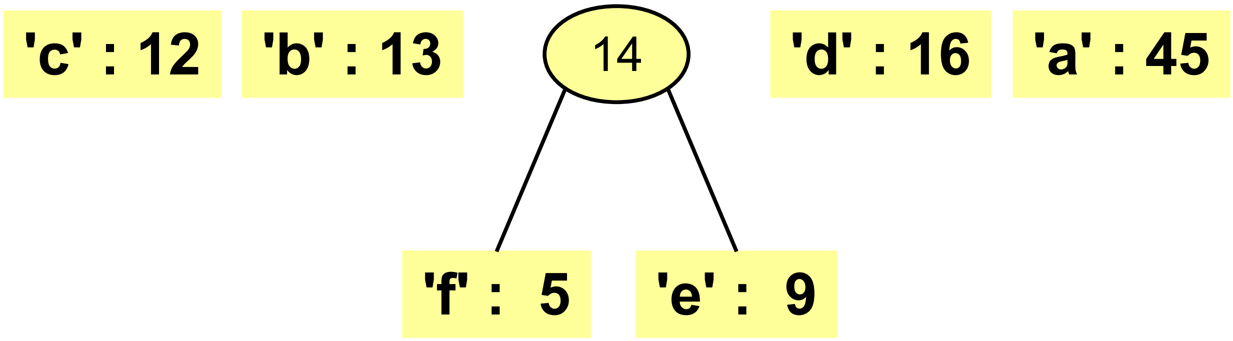
Costruzione del codice

- Passo 2: Rimuovere i due nodi con frequenze minori
- Passo 3: Collegarli ad un nodo padre etichettato con la frequenza combinata (sommata)



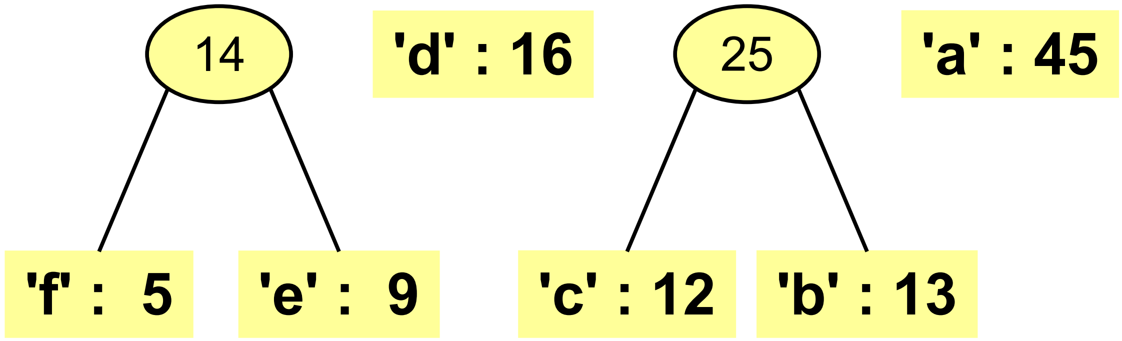
Costruzione del codice

- Passo 4: Aggiungere il nodo combinato alla lista, mantenendola ordinata in base alle frequenze



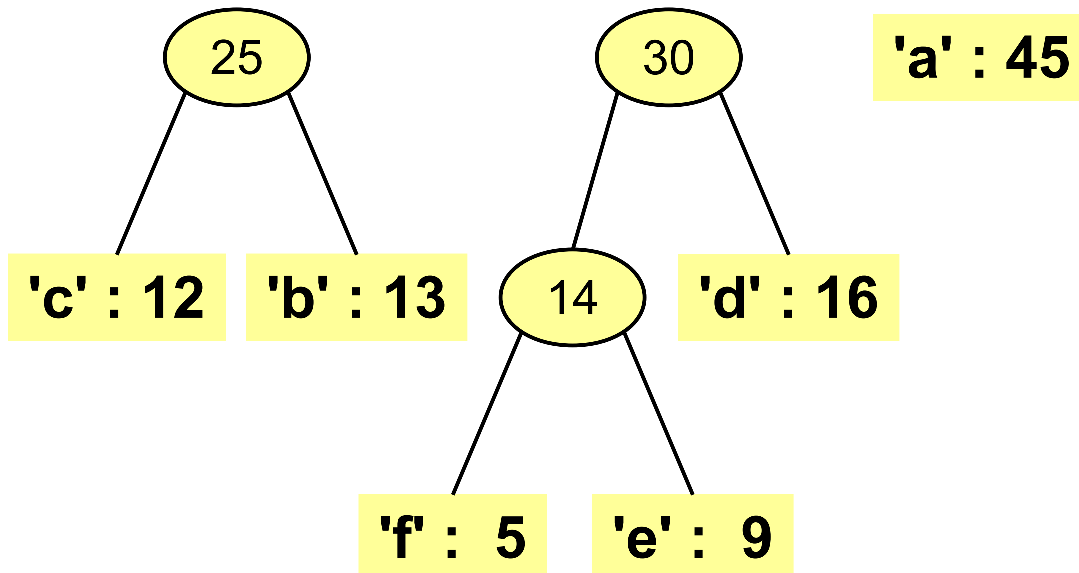
Costruzione del codice

- Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista



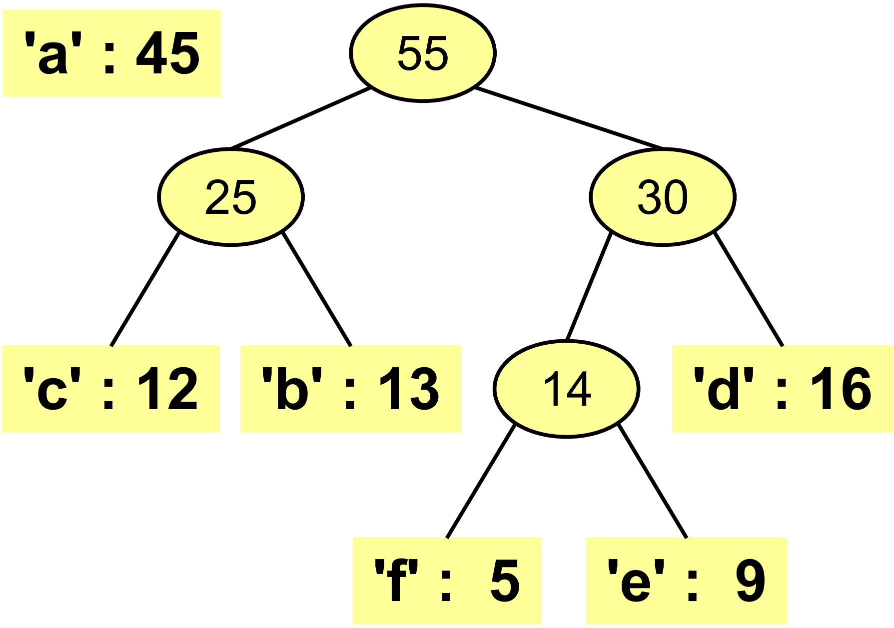
Costruzione del codice

- Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista



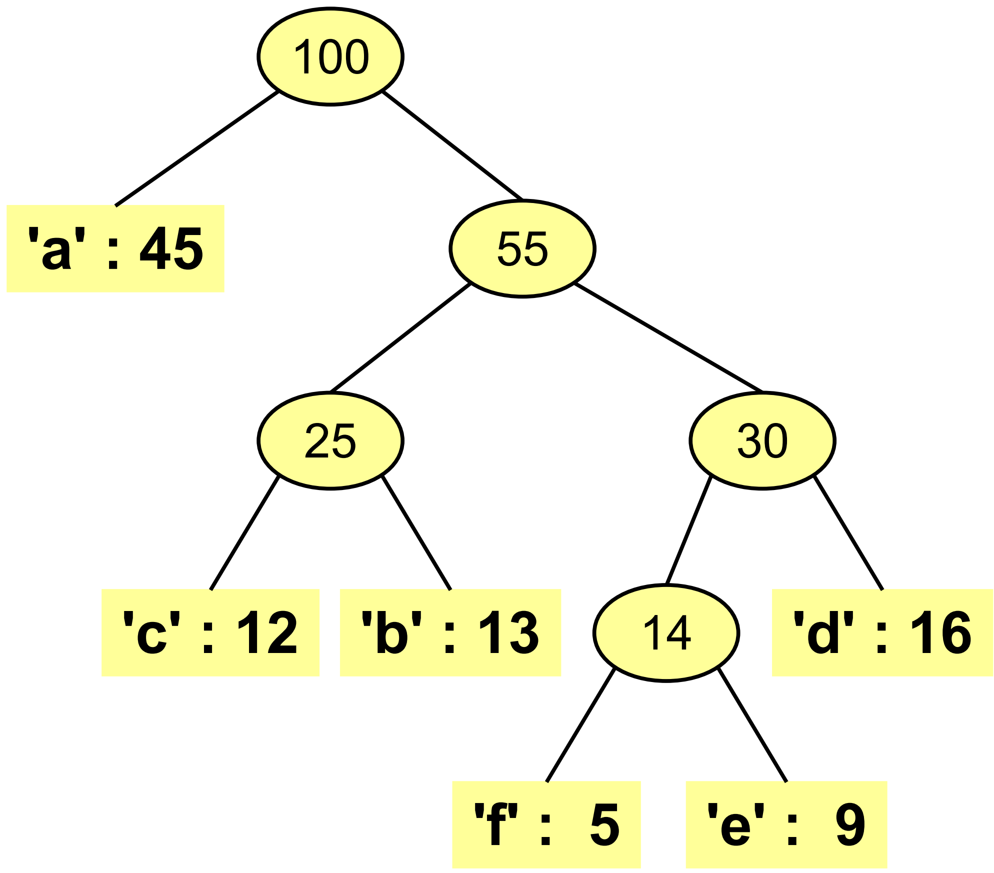
Costruzione del codice

- Ripetere i passi 2-4 fino a quando non resta un solo nodo nella lista



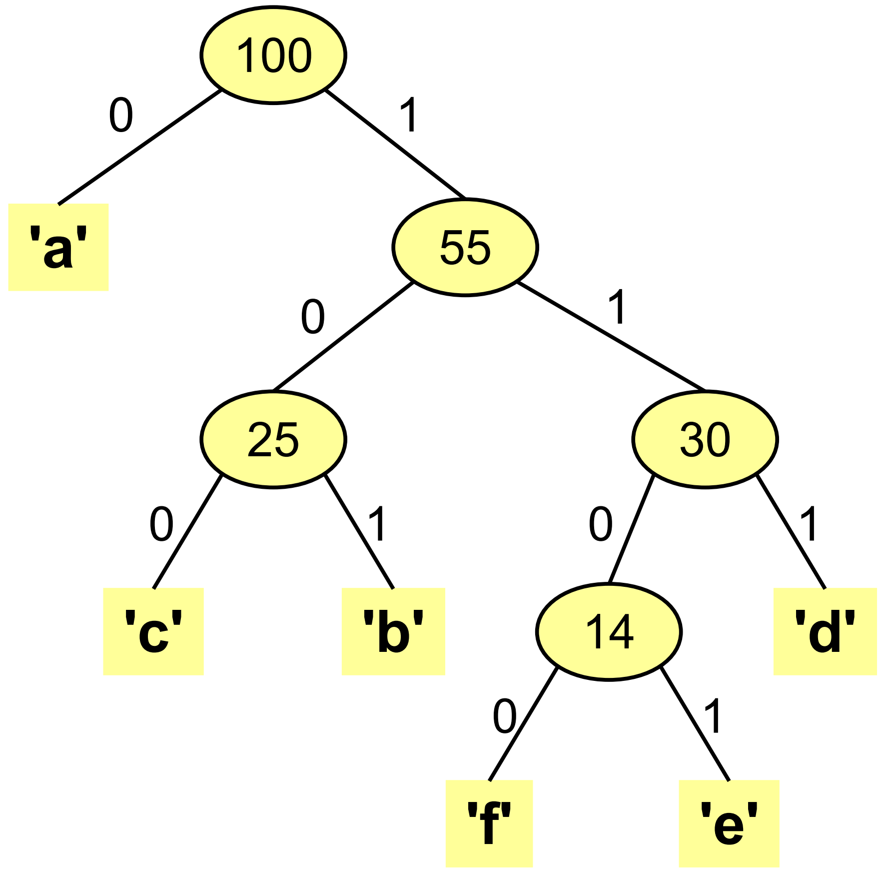
Costruzione del codice

- Al termine si etichettano gli archi dell'albero con 0 / 1



Costruzione del codice

- Al termine si etichettano gli archi dell'albero con 0 / 1



Algoritmo di Huffman

Costo O(*n log n*)

```
Huffman(real f[1..n], char c[1..n]) → Tree
    Q ← new MinPriorityQueue()
    integer i;
    for i ← 1 to n do
        z ← new TreeNode(f[i], c[i]);
        Q.insert(f[i], z);
    endfor
    for i ← 1 to n - 1 do
        z1 ← Q.findMin(); Q.deleteMin();
        z2 ← Q.findMin(); Q.deleteMin();
        z ← new TreeNode(z1.f + z2.f, '');
        z.left ← z1;
        z.right ← z2;
        Q.insert(z1.f + z2.f, z);
    endfor
    return Q.findMin();
```

insert(chiave, valore)

Struttura TreeNode:

f	**frequenza**
c	**carattere**
left	**figlio sinistro**
right	**figlio destro**

Algoritmi greedy

- Vantaggi
  - Semplici da programmare
  - Solitamente efficienti
  - In alcuni casi (quando è possibile dimostrare la proprietà di scelta greedy) danno la soluzione ottima
- Svantaggi
  - Non tutti i problemi ammettono una soluzione greedy
  - Quindi, in certi casi gli algoritmi greedy non possono essere usati se si vuole la soluzione ottima