

Tecniche Algoritmiche / 2

Algoritmi *Greedy*

Jocelyne Elias

<https://www.unibo.it/sitoweb/jocelyne.elias>

Moreno Marzolla

<https://www.moreno.marzolla.name/>

Dipartimento di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

Copyright © Alberto Montresor, Università di Trento, Italy

<http://cricca.disi.unitn.it/montresor/teaching/asd/>



Copyright © 2010—2016, 2021 Moreno Marzolla, Università di Bologna, Italy

<https://www.moreno.marzolla.name/teaching/ASD/>

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Introduzione

- Quando applicare la tecnica greedy?
 - Quando è possibile *dimostrare* che esiste una *scelta ingorda/greedy*
 - 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
 - in alcuni casi, soluzioni non ottime possono essere comunque interessanti

Schema di un algoritmo greedy generico

```
Greedy(insieme di candidati C) → soluzione
  S ← ∅
  while ( (not ottimo(S)) and (C ≠ ∅) ) do
    x ← seleziona(C)
    C ← C - {x}
    if (ammissibile(S ∪ {x})) then
      S ← S ∪ {x}
    endif
  endwhile
  if (ottimo(S)) then
    return S
  else
    errore ottimo non trovato
  endif
```

Ritorna *true* sse la
soluzione S è ottima

Estrae un candidato
dall'insieme C

Ritorna *true* sse la
soluzione candidata è una
soluzione ammissibile
(anche se non
necessariamente ottima)

Nota importante: questo **NON** è un algoritmo vero e proprio! questo è solo uno schema di algoritmo, che deve essere opportunamente modificato per adattarlo allo specifico problema da risolvere

Problema del resto

Problema del resto

- Input

- Un insieme di n tagli di monete/banconote $T[1..n]$
 - Disponiamo un numero infinito di monete di ciascun taglio
- 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

- Esempi:

$T[] = \{50, 20, 10, 5, 2, 1\}$, $R = 78 \rightarrow 5$ pezzi: $50+20+5+2+1$

$T[] = \{50, 20, 10, 5, 2, 1\}$, $R = 19 \rightarrow 4$ pezzi: $10+5+2+2$

Algoritmo greedy per il resto

- Insieme dei candidati C
 - Insieme dei tagli di monete a disposizione
- Soluzione S
 - Insieme delle monete da restituire
- **ottimo(S)**
 - *true* se la somma dei valori in S è uguale al resto
- **ammissibile(S)**
 - *true* se la somma dei valori in S è minore o uguale al resto
- **seleziona(C)**
 - eroga una moneta del massimo taglio minore o uguale al resto ancora da erogare
 - Nota: in questo algoritmo non bisogna modificare l'insieme C

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

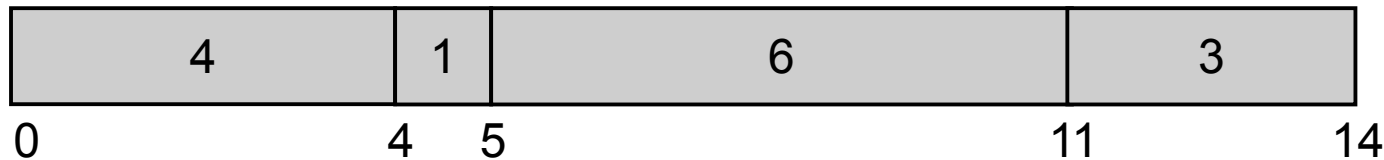
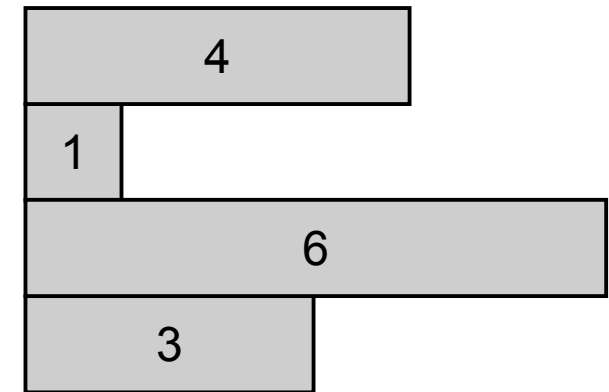
- I sistemi monetari per i quali l'algoritmo greedy fornisce la soluzione ottima si chiamano **sistemi monetari canonici**
 - Xuan Cai (2009). "**Canonical Coin Systems for CHANGE-MAKING Problems**". Proc. Ninth Int. Conf. on Hybrid Intelligent Systems 1: 499–504. doi:10.1109/HIS.2009.103.
- L'algoritmo 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 approccio diverso che produce 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, \dots, p_n
- Ogni job p_i ha un tempo di esecuzione $t[i]$
- Minimizzare il **tempo medio di completamento**



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



$$(1+4+8+14)/4 = 27/4$$

Algoritmo greedy di scheduling

- Siano p_1, p_2, \dots, 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

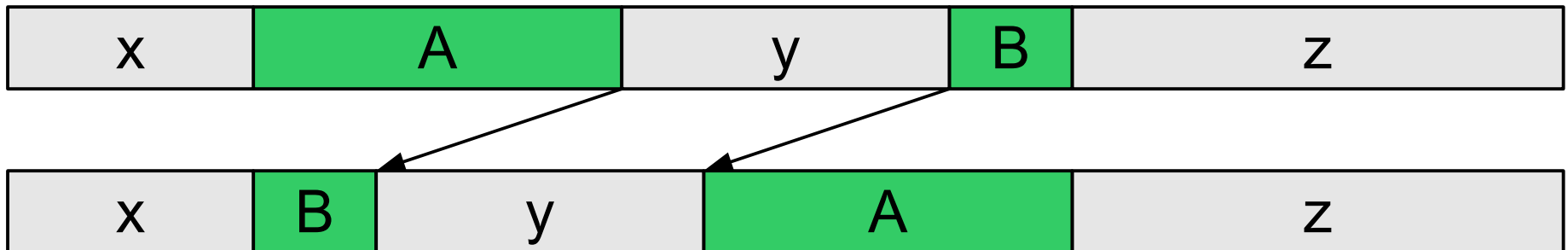
Algoritmo greedy per lo scheduling

Shortest-job-first

- Insieme dei candidati C
 - Job da schedulare, inizialmente $\{ p_1, p_2, \dots p_n \}$
- Soluzione S
 - Ordine dei job da schedulare: $p_{i1}, p_{i2}, \dots p_{in}$
- **ottimo(S)**
 - True se e solo se l'insieme S contiene tutti i job
- **ammissibile(S)**
 - Restituisce sempre true
- **seleziona(C)**
 - Sceglie il job di durata minima in C

Dimostrazione di ottimalità della scelta Greedy

- 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 è minore del tempo di completamento di A nella prima soluzione
 - Il tempo di completamento dei job in y si riduce

Problema della compressione (alberi di Huffman)

Problema della compressione

- Rappresentare i dati in modo efficiente
 - Impiegare il numero minore di bit per la rappresentazione dei dati
 - Scopo: risparmio spazio su disco e tempo di trasferimento su un canale di trasmissione
- Una possibile tecnica di compressione: *codifica di caratteri*
 - Tramite *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 ”

Codici di Huffman

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

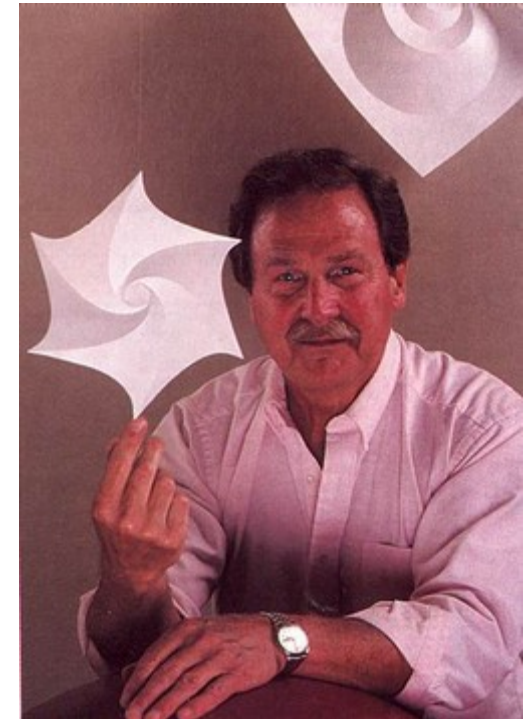
Codici di Huffman

- Codifica a lunghezza variabile
 - Caratteri: **'a'** **'b'** **'c'** **'d'** **'e'** **'f'**
 - Codifica: **0** **101** **100** **111** **1101** **1100**
 - Dimensione totale:
 $(0.45 \times 1 + 0.13 \times 3 + 0.12 \times 3 + 0.16 \times 3 + 0.09 \times 4 + 0.05 \times 4) \times N = 2.24N$
- Codice “a prefisso” (“senza prefissi”):
 - Nessun codice è un prefisso di un altro codice
 - Condizione sufficiente per permettere la decodifica
- Esempio: **addaabca**
 - $0 \cdot 111 \cdot 111 \cdot 0 \cdot 0 \cdot 101 \cdot 100 \cdot 0$

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

Codici di Huffman

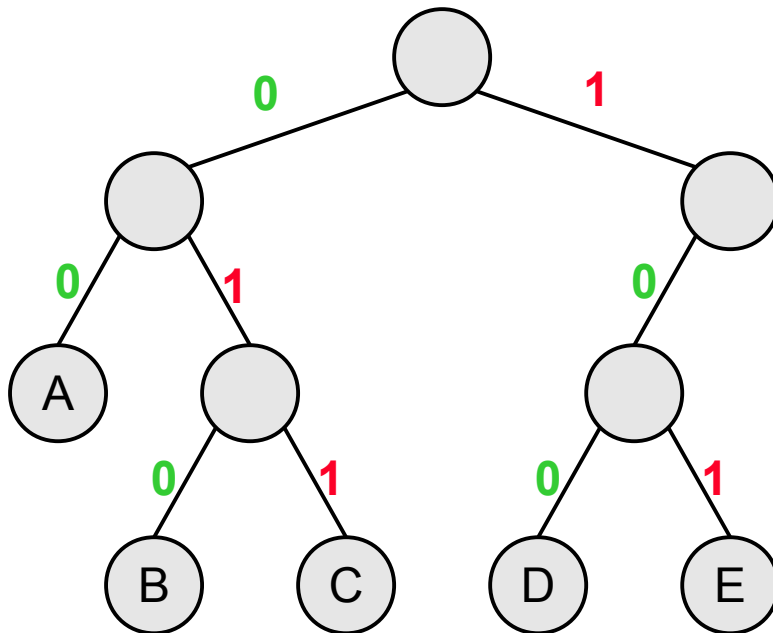
- Definire un algoritmo per la codifica è il tema dei lucidi seguenti
- Huffman, D.A., "*A Method for the Construction of Minimum-Redundancy Codes*," Proc. of the IRE, vol. 40, no. 9, pp. 1098—1101, Sept. 1952, doi: 10.1109/JRPROC.1952.273898
 - Algoritmo ottimo per costruire codici prefissi



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



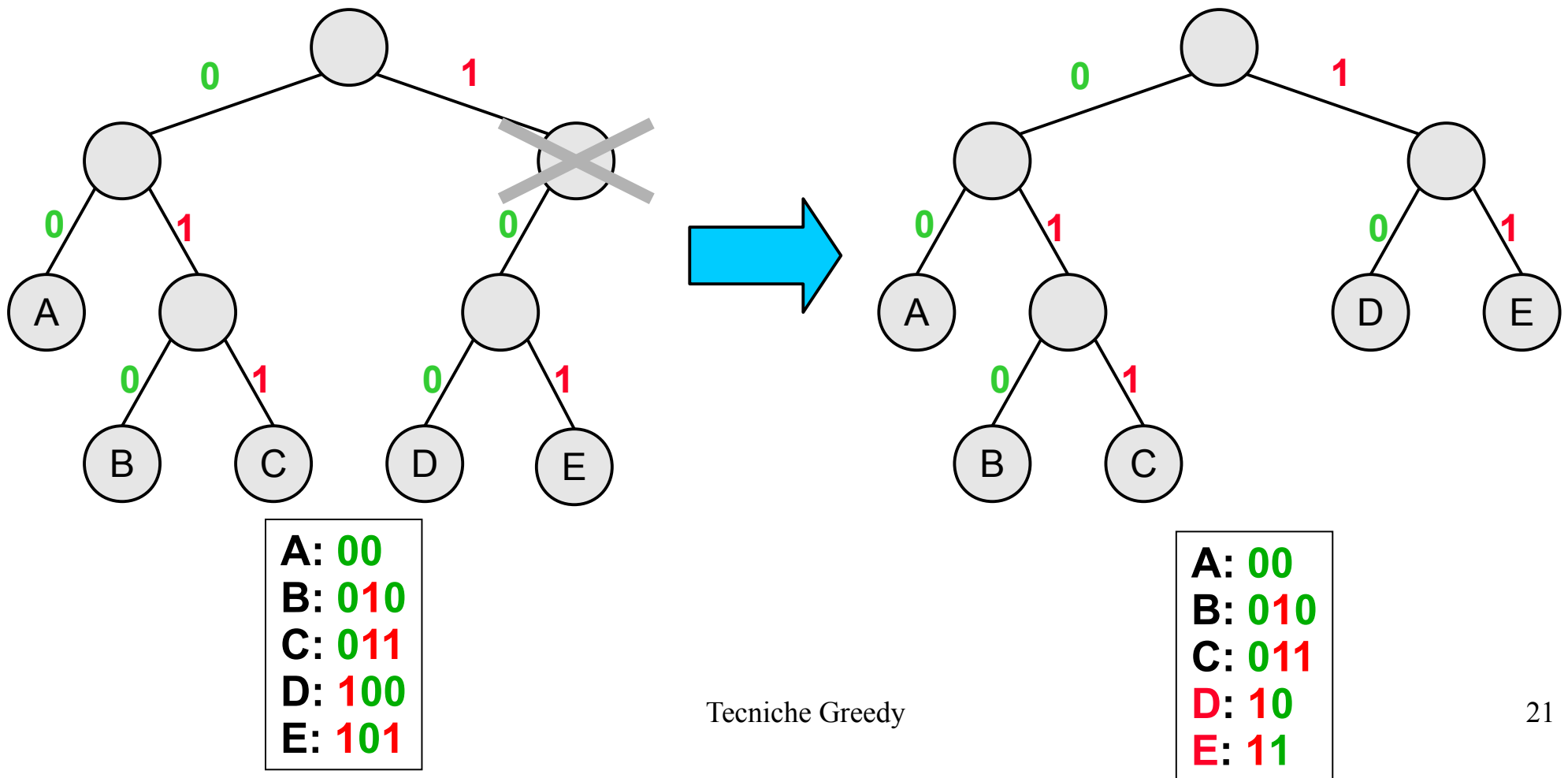
A: 00
B: 010
C: 011
D: 100
E: 101

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



Definizione formale del problema

- Definizione: **codice ottimo**
 - Dato un file F , una funzione di codifica f è ottima per F se non esiste un'altra funzione di codifica tramite la quale F possa essere compresso in un numero inferiore di bit.
- Nota:
 - La funzione di codifica ottima dipende dal file
 - Possono esistere più funzioni di codifica ottima

Algoritmo di Huffman

- Principio del codice di Huffman
 - Minimizzare la lunghezza delle codifiche 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 calcolano le frequenze di ciascun carattere
 - Si costruisce il codice
 - Si rappresenta il file tramite il codice
 - Si aggiunge al file una rappresentazione del codice

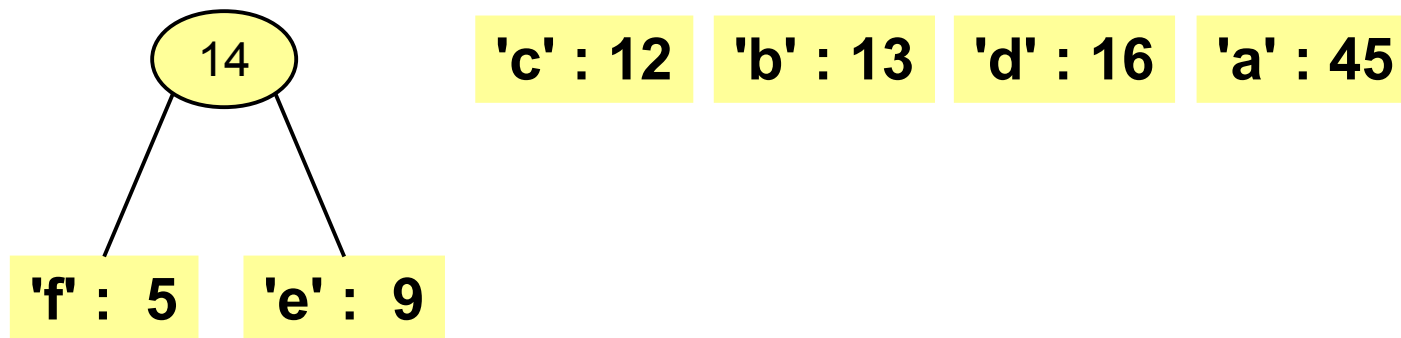
Costruzione del codice

- **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

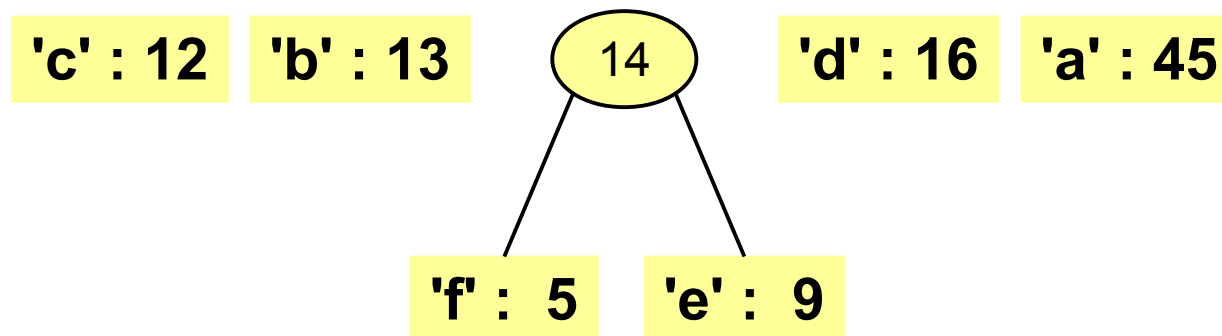
Costruzione del codice

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



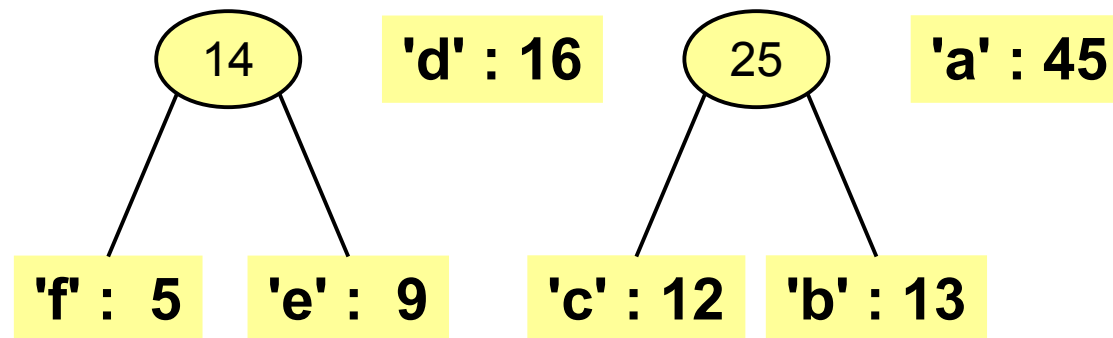
Costruzione del codice

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



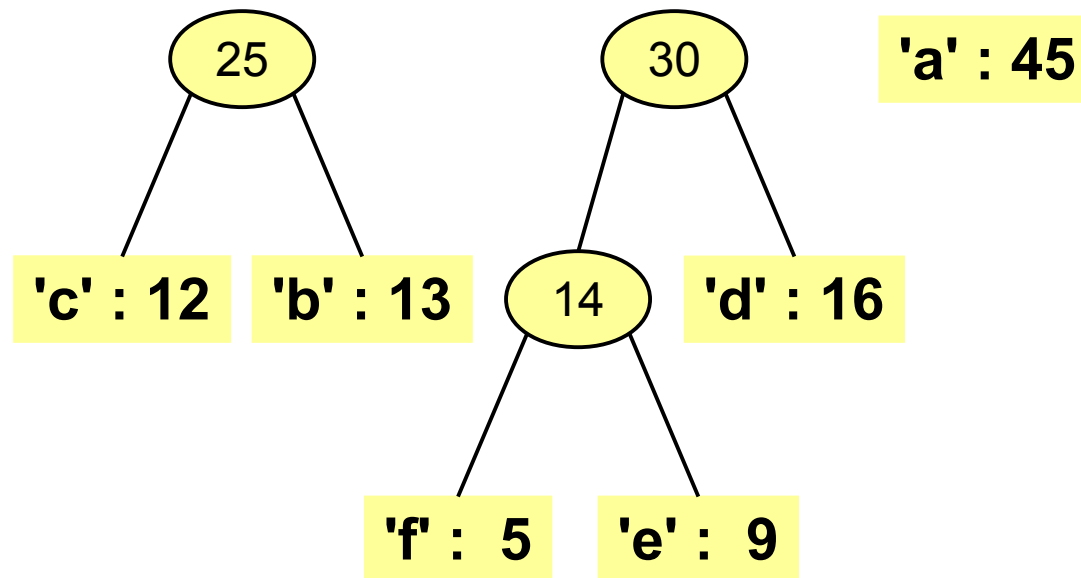
Costruzione del codice

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



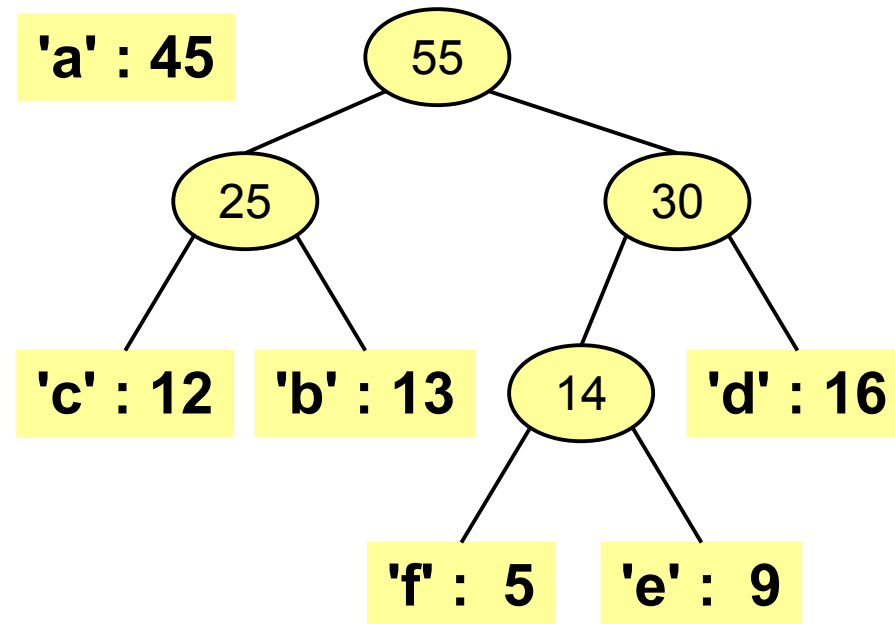
Costruzione del codice

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



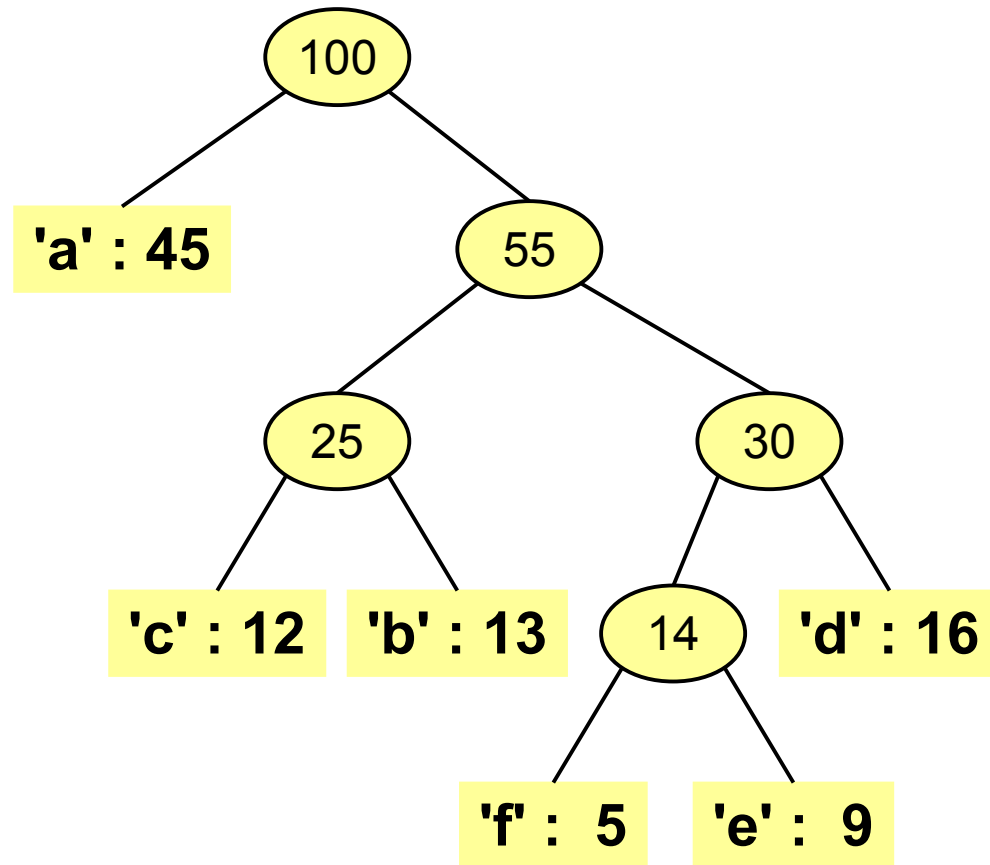
Costruzione del codice

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



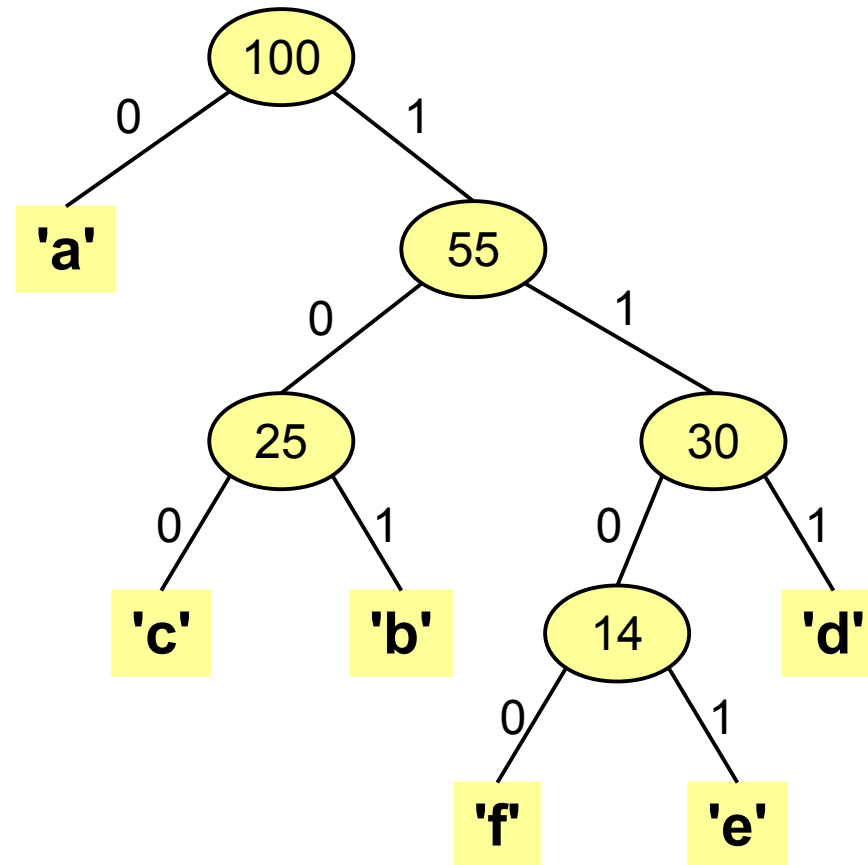
Costruzione del codice

- 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

```
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();
```

Costo $\Theta(n \log n)$

insert(chiave, valore)

Algoritmo Greedy

Struttura TreeNode:

<i>f</i>	frequenza
<i>c</i>	carattere
<i>left</i>	figlio sinistro
<i>right</i>	figlio destro

Algoritmi greedy

- **Vantaggi**

- Semplici da programmare
- Solitamente efficienti
- Quando è possibile dimostrare la proprietà di scelta greedy danno la soluzione ottima
- La soluzione sub-ottima può essere accettabile
-

- **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
 - Ma possono essere comunque applicati se ci si accontenta di una soluzione non necessariamente ottima