Codici di Huffman

I *codici di Huffman* vengono ampiamente usati nella compressione dei dati (pkzip, jpeg, mp3). Normalmente permettono un risparmio compreso tra il 20% ed il 90% secondo il tipo di file.

Sulla base delle frequenze con cui ogni carattere appare nel file, l'algoritmo di Huffman trova un *codice ottimo* — ossia un modo ottimale di associare ad ogni carattere una sequenza di bit detta *parola codice*.

Sia dato un file di 120 caratteri con frequenze:

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5

Usando un codice a lunghezza fissa occorrono 3 bit per rappresentare 6 caratteri. Ad esempio

carattere	a	b	c	d	e	f
cod. fisso	000	001	010	011	100	101

Per codificare il file occorrono $120 \times 3 = 360$ bit.

Possiamo fare meglio con un codice a lunghezza variabile che assegni codici più corti ai caratteri più frequenti. Ad esempio con il codice

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. var.	0	101	100	111	1101	1100

Bastano $57 \times 1 + 49 \times 3 + 14 \times 4 = 260$ bit.

Codici prefissi

Un *codice prefisso* è un codice in cui nessuna parola codice è prefisso (parte iniziale) di un'altra.

Ogni codice a lunghezza fissa è ovviamente prefisso.

Ma anche il codice a lunghezza variabile che abbiamo appena visto è un codice prefisso.

Codifica e decodifica sono particolarmente semplici con i codici prefissi.

Con il codice prefisso

carattere	a	b	c	d	e	f
cod. var.	0	101	100	111	1101	1100

La codifica della stringa **abc** è **0·101·100**.

La decodifica è pure semplice.

Siccome nessuna parola codice è prefisso di un'altra, la prima parola codice del file codificato risulta univocamente determinata.

Per la decodifica basta quindi:

1. individuare la prima parola codice del file codificato,
2. tradurla nel carattere originale e aggiungere tale carattere al file decodificato,
3. rimuovere la parola codice dal file codificato,
4. ripetere l'operazione per i successivi caratteri.

Ad esempio con il codice

carattere	a	b	c	d	e	f
cod. var.	0	101	100	111	1101	1100

la suddivisione in parole codice della stringa di bit **001011101** è **0·0·101·1101** a cui corrisponde la stringa **aabe**.

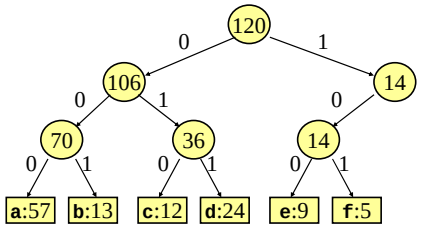
Per facilitare la suddivisione del file codificato in parole codice è comodo rappresentare il codice con un albero binario.

7

Ad esempio il codice a lunghezza fissa

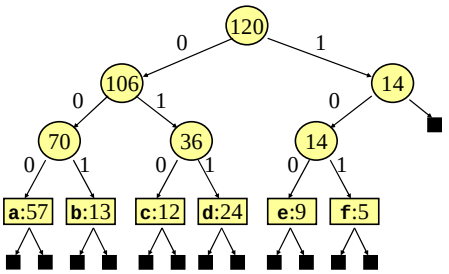
carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. fisso	000	001	010	011	100	101

ha la seguente rappresentazione ad albero



8

In realtà, come albero binario, la rappresentazione è



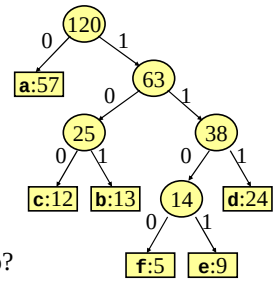
Noi trascureremo le foglie e chiameremo foglie i nodi interni senza figli.

9

Il codice a lunghezza variabile

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. var.	0	101	100	111	1101	1100

ha rappresentazione



Importa che si tratti di un codice prefisso?

10

La *lunghezza in bit del file codificato* con il codice rappresentato da un albero T è:

$$B(T) = \sum_{c \in C} f_c d_T(c)$$

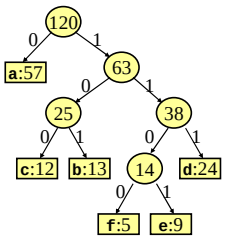
- C è l'alfabeto;
- f_c è la frequenza del carattere c e
- $d_T(c)$ è la profondità della foglia che rappresenta il carattere c nell'albero T.

Nota: assumiamo che l'alfabeto C contenga almeno due caratteri. In caso contrario basta un numero per rappresentare il file: la sua lunghezza.

11

La lunghezza in bit del file codificato è anche:

$$B(T) = \sum_{x \text{ nodo interno}} f[x]$$

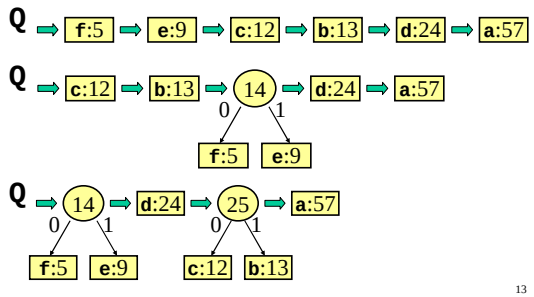


in cui la sommatoria è estesa alle frequenze $f[x]$ di tutti i nodi interni x dell'albero T.

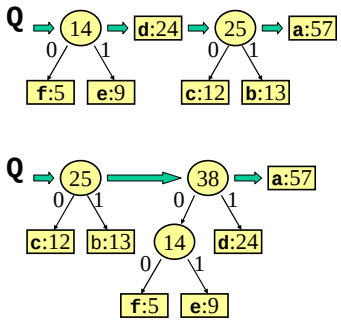
12

Costruzione dell'albero di Huffman:

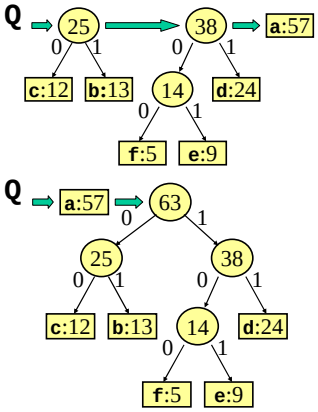
carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5



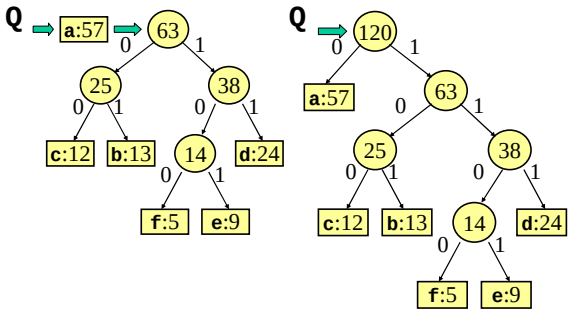
13



14



15



16

Implementazione dell'algoritmo goloso di Huffman.

```
Huffman(c, f, n)
  Q ← ϕ                          ▷ coda con priorità
  for i ← 1 to n do
     Push(Q, nodo(f_i, c_i))
  for j ← n downto 2 do
     x ← ExtractMin(Q), y ← ExtractMin(Q)
     Push(Q, nodo(x,y))
  return ExtractMin(Q)
```

Q è una coda con priorità, ***nodo(f, c)*** è il costruttore di un nodo foglia e ***nodo(x, y)*** è il costruttore di un nodo interno.

17

Assumendo che la coda ***Q*** venga realizzata con un heap, le operazioni ***Insert*** ed ***ExtractMin*** richiedono tempo $O(\log n)$.

Pertanto l'intero algoritmo richiede tempo $O(n \log n)$ (dove n è il numero di caratteri dell'alfabeto).

18

L'algoritmo è *goloso* perché ad ogni passo costruisce il nodo interno avente frequenza minima possibile. Ricordiamo infatti che

$$B(T) = \sum_{x \text{ nodo interno}} f[x]$$

Siamo sicuri che in questo modo otteniamo sempre un codice ottimo?

19

Elementi della strategia golosa

Ci sono due ingredienti comuni a molti problemi risolvibili con la strategia golosa:

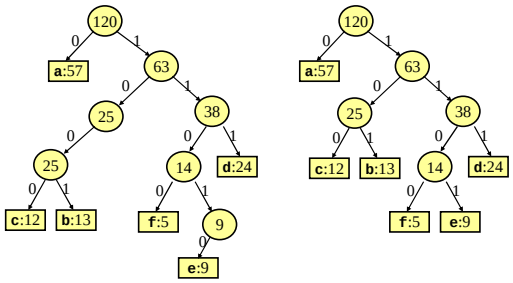
Proprietà della scelta golosa: La scelta ottima localmente (golosa) non pregiudica la possibilità di arrivare ad una soluzione globalmente ottima.

Sottostruttura ottima: Ogni soluzione ottima non elementare si compone di soluzioni ottime di sottoproblemi.

20

Osservazione: Se ***T*** è ottimo allora è *completo*.

Ogni nodo interno di ***T*** ha due figli (altrimenti togliendo il nodo si otterrebbe un codice migliore).



21

Lemma (*proprietà della scelta golosa*)
Siano ***a*** e ***b*** due caratteri di ***C*** aventi frequenze f_a ed f_b minime.
Esiste un codice prefisso ottimo in cui le parole codice di ***a*** e ***b*** hanno la stessa lunghezza e differiscono soltanto per l'ultimo bit (e quindi, nell'albero del codice, le foglie associate ad ***a*** e ***b*** sono figlie dello stesso nodo interno, cioè sorelle).

Attenzione: Il Lemma **non** dice che ciò è vero per ogni codice prefisso ottimo e **tanto meno** che se ciò è vero il codice è ottimo!!!!
Dice **solo** che ciò è vero per **almeno** un codice ottimo.

22

Sia ***T*** ottimo. Quindi in ***T*** esistono due foglie a profondità massima che sono sorelle.

Siano ***c*** e ***d*** i caratteri corrispondenti a tali foglie.

Mostriamo che scambiando ***c*** e ***d*** con ***a*** e ***b*** il codice rimane ottimo.

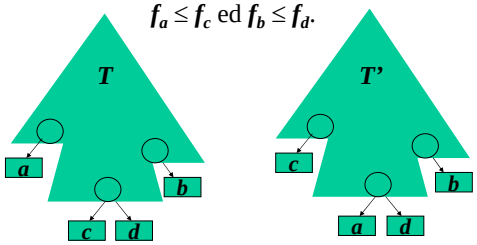
Possiamo supporre $f_c \le f_d$ e $f_a \le f_b$.

a e ***b*** sono due caratteri con frequenza minima in assoluto e quindi $f_a \le f_c$ ed $f_b \le f_d$.

23

Sia ***T'*** l'albero che si ottiene da ***T*** scambiando le foglia associata al carattere ***c*** con quella associata al carattere ***a*** (e ricalcolando le frequenze dei nodi interni).

$f_a \le f_c$ ed $f_b \le f_d$.



24

Allora:

$$\begin{aligned} B(T) - B(T') &= \sum_{e \in C} f_e d_T(e) - \sum_{e \in C} f_e d_{T'}(e) \\ &= f_a d_T(a) + f_c d_T(c) - f_a d_{T'}(a) - f_c d_{T'}(c) \\ &= f_a d_T(a) + f_c d_T(c) - f_a d_T(c) - f_c d_T(a) \\ &= [f_c - f_a][d_T(c) - d_T(a)] \\ &\geq 0 \end{aligned}$$

Siccome ***T*** è ottimo ***B(T)*** = ***B(T')*** e quindi anche ***T'*** è ottimo.

Allo stesso modo, scambiando le foglie dei caratteri ***d*** e ***b***, si ottiene un albero ottimo ***T''*** in cui ***a*** e ***b*** sono associati a due foglie sorelle.

25

Lemma (*proprietà della sottostruttura ottima*)
Sia ***C*** un alfabeto e siano ***a*** ed ***b*** i caratteri con frequenza minima. Sia ***C'*** = ***C*** \{***a,b***}∪{***c***} l'alfabeto ottenuto sostituendo a {***a***, ***b***} un unico carattere nuovo ***c*** (con frequenza $f_c = f_a + f_b$). Sia ***T'*** l'albero di un codice prefisso ottimo per ***C'***.

Allora l'albero ***T*** ottenuto da ***T'*** sostituendo la foglia corrispondente a ***c*** con un nodo interno, avente come figli le foglie ***a*** ed ***b*** rappresenta un codice prefisso ottimo per l'alfabeto ***C.***

26

$$\begin{aligned} B(T) &= B(T') + f_a d_T(a) + f_b d_T(b) - f_c d_{T'}(c) \\ &= B(T') + (f_a + f_b)(d_{T'}(c) + 1) - (f_a + f_b) d_{T'}(c) \\ &= B(T') + f_a + f_b \end{aligned}$$

Supponiamo, per assurdo, che esista un albero ***S*** per ***C*** tale che ***B(S)*** < ***B(T)***.

Per il lemma precedente possiamo assumere che le foglie corrispondenti ad ***a*** e ***b*** siano sorelle, figlie di un nodo ***z***.

27

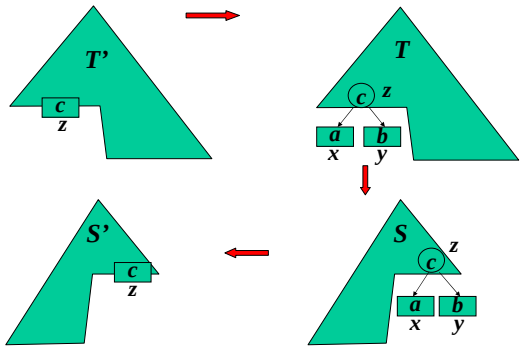
Consideriamo l'albero ***S'*** per ***C'*** ottenuto da ***S*** facendo diventare ***z*** una foglia, corrispondente al carattere ***c***, (con frequenza $f_c = f_a + f_b$).

Allora, come in precedenza:

$$\begin{aligned} B(S') &= B(S) - (f_a + f_b) \\ &< B(T) - (f_a + f_b) \\ &= B(T') \end{aligned}$$

Questo contraddice l'ottimalità di ***T'***. Assurdo.

28



29

Teorema
L'algoritmo di Huffman produce un codice prefisso ottimo.

```
Huffman(c, f, n)
  Q ← ϕ                          ▷ coda con priorità
  for i ← 1 to n do
     Push(Q, nodo(f_i, c_i))
  for j ← n downto 2 do
     x ← ExtractMin(Q), y ← ExtractMin(Q)
     Insert(Q,nodo(x,y))
  return ExtractMin(Q)
```

Si dimostra per induzione sul numero di caratteri dell'alfabeto (lunghezza della coda) usando i due lemmi precedenti.

30

Esercizio 5. Compressione Universale?
Dimostrare non esiste una schema di compressione in grado di ridurre la lunghezza di ogni file di caratteri di 8 bit (ci sarà sempre un file che la cui codifica non e' più corta del file originale).
(*Suggerimento*: confrontare il numero dei file con il numero dei file codificati.)

31

Supponiamo per assurdo che un tale schema esista.

Ci sono 2^8 file diversi di un solo carattere (8 bit).
Le sequenze di bit di lunghezza minore di *8* sono 2^8-1.
Ognuno dei file di un carattere corrisponde a qualcuna di queste sequenze.

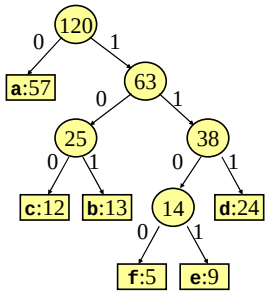
Dunque, per il principio dei buchi di colombaia, ci sono due file diversi con codifica identica.

ASSURDO!!!!

32

Esercizio 6.
Sia $C = \{c_1, ..., c_n\}$ un insieme di caratteri e siano $f_1, ..., f_n$ le loro frequenze in un file.
Mostrare come si possa rappresentare ogni codice prefisso ottimo per C con una sequenza di
$2n - 1 + n\lceil \log n \rceil$ bits.
Suggerimento: usare $2n - 1$ bit per rappresentare la struttura dell'albero del codice ed $n\lceil \log n \rceil$ bits per elencare i caratteri nell'ordine in cui compaiono nelle foglie (usando il codice del file non compresso).

33

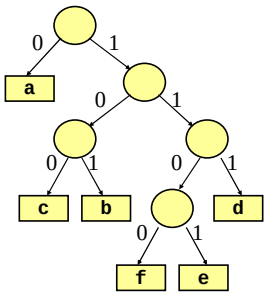


Parti dalla radice e ricorsivamente:
- se è un nodo int. metti 0
  - scendi sul figlio sx;
  - scendi sul figlio dx.
- se è una foglia metti 1

01001100111

34

01001100111



Crea la radice e ricorsiv. :
- se incontri uno 0
  - crea il figlio sx e scendi su tale figlio;
  - crea il figlio dx e scendi su tale figlio
- se incontri un 1, stop.

35