Codice di Huffman

1 – IL PROBLEMA

2 – GLI ALGORITMI E LE STRUTTURE DATI IMPLEMENTATE (con la loro complessità asintotica)

3 – UN ESEMPIO PRATICO

4 – RISULTATI SPERIMENTALI

1-IL PROBLEMA

- Descrizione del problema :

- come possiamo trasmettere la stessa quantità di INFORMAZIONE trasmettendo il numero minore possibile di BIT ??

- E' dato un file testuale (.txt) di n byte generici, siamo interessati a trasmetterlo utilizzando il numero minore possibile di bit, per sprecare meno risorse possibile. Siamo interessati cioè a creare una codifica diversa da quella standard, che possa comprimere l'informazione del nostro testo in un numero inferiore di byte !

- L'idea :

- Perchè non cercare di usare meno bit per I caratteri usati più spesso e più bit per I caratteri usati meno di frequente ? (di norma vengono usati la stessa quantità di bit per qualsiasi carattere, non tenendo conto della frequenza con cui ricorrono in un testo !)

- In teoria : se un carattere è usato molto più spesso in un testo, e questa è un'informazione che già conosco a priori, prima di leggere ogni carattere, posso già aspettarmi una probabilità più

alta di incontrarlo, ma se lo trasmetto sprecando gli stessi bit di uno poco probabile, sto perdendo informazione

- Cioè : i caratteri usati più frequentemente hanno un contenuto informativo inferiore a quelli usati di rado !

- L’approccio alla soluzione :

- Creare tramite l’analisi dei caratteri più usati, una codifica in grado di comprimere la soluzione, generando una tabella che poi ci permetta di tornare indietro al testo originale, senza perdita di informazione

- Questa tabella conterrà le coppie CARATTERE/CODIFICA che abbiamo generato nel nostro algoritmo

2 – GLI ALGORITMI E LE STRUTTURE DATI IMPLEMENTATE

- (step 1) Individuare la frequenza di ogni carattere in un testo

frequenza : numero di volte che un carattere si ripete in un file

STRUTTURA DATI UTILIZZATA ⟶ HASH TABLE (char / int)

pseudo codice :

```
while ( testo_has_char() )
    if ( carattere in key_hash_table)
        hash_table[carattere] ++
    else
        hash_table[carattere] = 1
```

L'algoritmo scorre riga per riga tutto il testo ed inserisce ogni nuovo carattere come chiave di una hash table, settando il valore ad 1 (frequenza), se lo incontra nuovamente, incrementa il valore associato alla frequenza di 1

COMPLESSITA' O(n) !! ---> viene effettuata per n volte (numero di caratteri nel testo) , un'operazione a costo costante (inserimento/aggiornamento chiave in una tabella hash)

nel codice java è stata implementata una hash_table char/mutableInt dove mutableInt è una classe custom che permette di incrementare il 'contatore' più efficientemente, è stata implementata così solo a scopo di efficienza

RISULTATO (di esempio) :

A	B	C	D	E	F
10	2	5	14	23	7

Alla fine del passo 1 avremo una tabella hash dove sono presenti tutti i caratteri incontrati con le loro relative frequenze, saranno di futuro interesse nei prossimi step

-(Step 2) Ordinare i caratteri (ordine decrescente)

Ordinare i caratteri della hash table a seconda della frequenza relativa

STRUTTURA DATI UTILIZZATA ⟶ CODA DI PRIORITA’

pseudo codice :

```
for chiave in hash_map
    data = ( hash_map[key] , key )
    key coda.add(( data ))
```

L’algoritmo scorre tra le chiavi della tabella hash costruito sopra , ed aggiunge alla coda la coppia (char, frequenza)

COMPLESSITA’ O(n log n) !! ---> viene effettuara per n volte (numero di chiavi nella tabella hash) un’operazione a costo logaritmico (inserimento di un elemento in una coda di priorità)

Nel codice java si è implementata una Coda di Priorità di Nodi (che saranno poi i Nodi dell’albero binario, all’interno di ogni nodo c’è un campo data dove ci sono, per ogni nodo i rispettivi valori di frequenza e carattere, la priorità è scelta in base alla frequenza nel campo data di ogni nodo (è stato implementato un nodo comparator per stabilire la priorità dei nodi)!

alla fine del passo 2 avremo una coda di Nodi da cui poter estrarre in maniera ordinata, uno alla volta i nodi, la scelta è ricaduta su una coda di priorità perchè non basta un semplice ordinamento iniziale : dei nuovi nodi saranno inseriti nella coda in punti intermedi dei prossimi algoritmi !

-(Step 3) Creare una codifica valida per il testo

STRUTTURA DATI UTILIZZATA ⟶ ALBERO BINARIO

pseudo codice :

```
n = numero di caratteri
C = coda di priorità con i nodi ordinati
for i = 1 to n
        creo nuovo nodo N
        N.left = C.estrai_min()
        N.right = C.estrai_min()
        N.data = N.right.freq + N.left.freq
        C.inserisci(N)
```

L'algoritmo esegue n volte un ciclo for dove vengono estratti i due nodi con valore di frequenza minimo, vengono sommati in un nuovo nodo, che viene re-inserito nella coda di priorità

COMPLESSITA' O(n log n) !! viene eseguita per n volte (numero di caratteri da codificare un'operazione a costo logaritmico (inserire un elemento in una coda di priorità)

Nel codice i nodi sono stati implementati con un campo info, dove all'interno ci sono sia il carattere che la frequnza del carattere, associati a quel nodo ora, come unico elemento della coda sarà rimasta la radice dell'albero binario che come foglie avrà i nodi carattere/frequenza del testo e nei blocchi intermedi un carattere generico (nel codice java '£') e un numero che rappresenta la somma delle frequenze dei nodi sottostanti

! ! SCELTA GOLOSA UTILIZZATA NELL'ALGORITMO SOPRA ! !

La scelta golosa consiste nello scegliere ogni volta come coppia di nodi da sommare, i due nodi con frequenza minore presenti nella coda, l'utilità di questo algoritmo, parte centrale in tutta la codifica di huffman , è che costruisce un codice prefisso ,fondamentale per una codifica/decofica !

Codice prefisso : codifica in cui le parole codificate NON sono prefisso di nessun'altra parola codificata, ciò è fondamentale in fase di decodifica , altrimenti scorrendo i bit della codifica, potremmo confodere delle parole !

in particolare il codice prefisso è ottenuto raggiungendo ogni foglia dell'albero, partendo dalla radice, e aggiungendo alla codifica della foglia (inizialmente stringa vuota) 0 o 1 a seconda del percorso fatto dalla radice per arrivare alla foglia (ogni volta che si va a sinistra si aggiunge 1 , ogni volta che si va a destra 0)

pseudo codice (per scorrere l'albero ed ottenere la codifica) :

```
Ricorsione(root, encoding)
    if root = nodo foglia
        root.enc = encoding
    else
        Ricorsione(root.left, encoding+1)
        if root.right != NIL
            Ricorsione(root.right, encoding+0)
```

L'algoritmo ricorsivo aggiunge ad ogni chiamata ricorsiva 0 o 1 a seconda del nodo sinistro o destro dove è stata fatta la chiamata ricorsiva, nel caso base, quando arriva ad un nodo foglia, assegna l'encoding costruito dalle chiamate precedenti al nodo foglia

N.B nel codice java reale, questo codice ricorsivo popola una tabella hash con all'interno le coppie carattere/codifica e viceversa !

alla fine del passo 3 avremo un albero binario che ha come foglie i nodi con all'interno char/frequenza ed una codifica valida da utilizzare presente in una tabella hash con delle coppie CARATTERE/CODIFICA

-(step 4) Encoding del file di testo

STRUTTURA DATI UTILIZZATA ⟶ HASH TABLE (char / char)

pseudo codice :

```
for c in file_testo:
    char_encoded = hash_table.get(c)
    file_encode.write(char_encoded)
```

L'algoritmo scorre ogni carattere del testo , ricava dall'hash table ottenuta nello step sopra il carattere codificato e lo scrive su file

COMPLESSITA' O(n) !! Si esegue per n volte (numeri di caratteri in un testo) un operazione a costo costante (recupero di un valore in una tabella hash)

-(step 5) Decoding del file di testo codificato

STRUTTURA DATI UTILIZZATA ⟶ HASH TABLE (char / char)

pseudo codice :

```
for bit in file_encode:
    compare += bit
    if hash_table.get(compare)
        char_decoded = hash_table.get(compare)
        file_decode.write(char_decoded)
        compare = " "
```

COMPLESSITA' O(n) !! Si esegue per n volte (numeri di caratteri in un testo) un operazione a costo costante (recupero di un valore in una tabella hash)

Nei passi 4 e 5 si suppone di avere una hashtable con all'interno delle coppie carattere/codifica e viceversa, nel codice java sono costruite velocemente durante l'esecuzione del codice ricorsivo

3 - UN ESEMPIO PRATICO

Nella cartella huffman_code è presente una cartella esempi_testi , a scopo di esempio è riportato l'output ottenuto del file esempio1.txt, in particolare sono riportati le varie strutture dati popolate durante l'esecuzione dell'algoritmo

Nel file è presente solamenta la parola 'esempio'

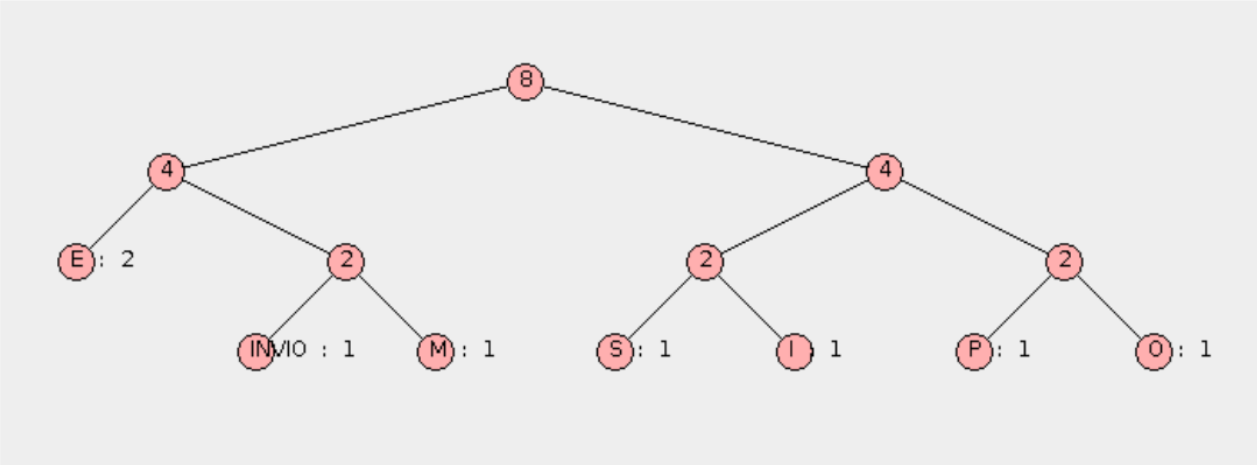
TABELLA HASH DELLE FREQUENZE OTTENUTA :

CARATTERE	FREQUENZA
P	1
S	1
E	2
I	1
'INVIO'	1
M	1
O	1

CODA DI PRIORITA' OTTENUTA :

NODO P PRIORITY 1
NODO O PRIORITY 1
NODO S PRIORITY 1
NODO INVIO PRIORITY 1
NODO I PRIORITY 1
NODO M PRIORITY 1
NODO E PRIORITY 2

ALBERO OTTENUTO :



CODIFICA OTTENUTA :

CODIFICA	CARATTERE
11	E
011	S
000	O
001	P
100	M
101	INVIO
010	I

4 – RISULTATI SPERIMENTALI

Tempi di esecuzione :

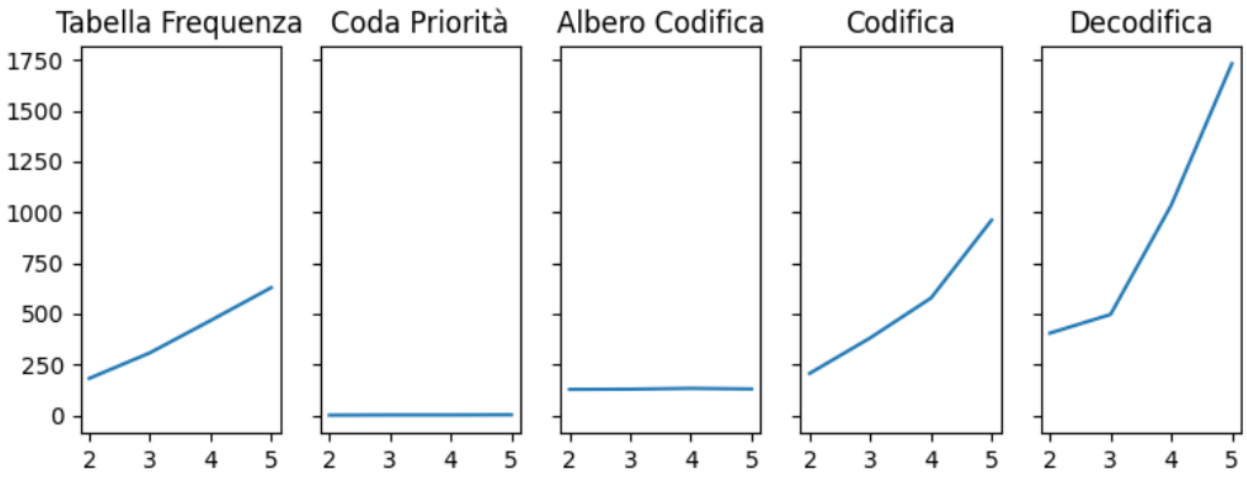
Nella cartella esempi_testi sono presenti 4 file utilizzati per i vari test

Considerando le dimensioni dei file :

nome file	esempio2	esempio3	esempio4	esempio5
dimensione(in MB)	1,9	3,8	8,7	17,4
dimensione compresso (in MB)	1,0	2,1	4,8	9,7
Fattore di Compressione	0,52	0,55	0,554	0,557

Le dimensioni dei file sono via via crescenti

Questi sono i grafici che visualizzano la crescita del tempo richiesto (in millisecondi) al crescere della dimensione del file di input (abbiamo crescite lineari o costanti , come ci aspettavamo dall'analisi asintotica della complessità)



- il costo della costruzione della tabella di hash cresce linearmente all'aumentare dei MB del file, ossia del numero di caratteri da scorrere
- La coda di priorità ha costo pressochè costante all'aumentare dei caratteri perchè il numero di nodi non aumenta : questo perchè all'interno della coda ci sono tanti nodi quanti sono i caratteri DISTINTI nel testo
- l'albero di codifica ha una crescita costante per lo stesso motivo della coda di priorità, il numero di nodi su cui si esegue l'algoritmo rimane pressochè costante all'aumentare della dimensione dei file
- La Codifica/Decodifica hanno una crescita lineare come ci aspettavamo, all'aumentare dei caratteri come nella costruzione della tabella hash, aumentano linearmente i rispettivi caratteri di codifica/decodifica