Codifica di Huffman relazione

- **Il problema:**

Il laboratorio in questione consiste nel creare una codifica di Huffman per i caratteri ASCII i quali vengono codificati in un byte ciascuno e, dal momento che la codifica ASCII utilizza 7 bit su 8 per identificare 128 caratteri, vi è uno spreco di spazio.

Quindi dopo aver compresso il file in input con una codifica di Huffman lo confronterò con il file in output, il quale conterrà solo uni e zeri, e calcolerò la percentuale di compressione (considerando che ogni 1 e 0 presente nel file di output occupa solo un bit di memoria).

- **Il codice:**

Ho svolto il laboratorio in Python dal momento che possiede molte funzioni integrate che permettono di lavorare con più facilità su file in input/output.

Come prima cosa ho definito la struttura *node*, che conterrà per ogni carattere, la sua codifica e la probabilità con il quale si presenta nel testo che andrò a leggere. A questo punto leggo il file (Lorem Ipsum) e memorizzo in un dizionario per ogni carattere il numero di volte che compare nel testo. Dopo aver ordinato il dizionario stampo su terminale quante volte compare ogni carattere e la sua probabilità.

```
Il carattere   compare 14610 volte con una probabilita' del: 14.61 %
Il carattere e compare 9560 volte con una probabilita' del: 9.56 %
Il carattere i compare 7890 volte con una probabilita' del: 7.89 %
Il carattere u compare 7600 volte con una probabilita' del: 7.6 %
Il carattere s compare 6910 volte con una probabilita' del: 6.91 %
Il carattere t compare 6490 volte con una probabilita' del: 6.49 %
Il carattere a compare 6240 volte con una probabilita' del: 6.24 %
Il carattere l compare 5030 volte con una probabilita' del: 5.03 %
Il carattere n compare 4860 volte con una probabilita' del: 4.86 %
Il carattere r compare 4360 volte con una probabilita' del: 4.36 %
Il carattere m compare 3610 volte con una probabilita' del: 3.61 %
Il carattere o compare 3440 volte con una probabilita' del: 3.44 %
Il carattere c compare 3210 volte con una probabilita' del: 3.21 %
Il carattere . compare 2280 volte con una probabilita' del: 2.28 %
Il carattere d compare 2090 volte con una probabilita' del: 2.09 %
Il carattere p compare 1980 volte con una probabilita' del: 1.98 %
Il carattere , compare 1970 volte con una probabilita' del: 1.97 %
Il carattere g compare 1170 volte con una probabilita' del: 1.17 %
Il carattere v compare 1130 volte con una probabilita' del: 1.13 %
Il carattere b compare 1040 volte con una probabilita' del: 1.04 %
Il carattere q compare 1000 volte con una probabilita' del: 1.0 %
Il carattere f compare 530 volte con una probabilita' del: 0.53 %
Il carattere h compare 470 volte con una probabilita' del: 0.47 %
Il carattere P compare 420 volte con una probabilita' del: 0.42 %
Il carattere N compare 280 volte con una probabilita' del: 0.28 %
Il carattere C compare 230 volte con una probabilita' del: 0.23 %
Il carattere S compare 210 volte con una probabilita' del: 0.21 %
Il carattere V compare 210 volte con una probabilita' del: 0.21 %
Il carattere D compare 190 volte con una probabilita' del: 0.19 %
Il carattere A compare 170 volte con una probabilita' del: 0.17 %
Il carattere I compare 140 volte con una probabilita' del: 0.14 %
Il carattere F compare 130 volte con una probabilita' del: 0.13 %
Il carattere M compare 130 volte con una probabilita' del: 0.13 %
Il carattere j compare 110 volte con una probabilita' del: 0.11 %
Il carattere E compare 90 volte con una probabilita' del: 0.09 %
Il carattere Q compare 60 volte con una probabilita' del: 0.06 %
Il carattere U compare 60 volte con una probabilita' del: 0.06 %
Il carattere y compare 50 volte con una probabilita' del: 0.05 %
Il carattere ; compare 30 volte con una probabilita' del: 0.03 %
Il carattere L compare 20 volte con una probabilita' del: 0.02 %
```

Ora, attraverso la funzione *shannonEntropy(),* calcolo l'entropia di Shannon con la seguente formula:

$$\sum_i p_i \log_2 \frac{1}{p_i}$$

Ottenendo che:

```
L'entropia di Shannon e': 4.25
```

E con la funzione *lunghezzaAttesa()* trovo la lunghezza attesa della codifica, calcolata nel seguente modo:

$$\sum_{x \in \mathcal{X}} p(x) L_C(x).$$

Ottenendo che:

```
La lunghezza attesa della codifica e': 4.28
```

Adesso passo a costruire l'albero di Huffman tramite la funzione *huffmanTree()* che crea una lista contenente un nodo per ogni carattere, e fino a quando non rimane un solo nodo nella lista prende i due con probabilità minore, li assegna come codifica un 1 o uno 0, li rende figli di un nodo padre contenente come probabilità la somma dei figli e infine rimuove quest'ultimi dalla lista e aggiunge il corrispettivo nodo padre. Ad ogni iterazione devo riordinare la lista in modo da avere nelle prime due posizioni sempre i nodi con probabilità minore.

Infine, la funzione *rewriteFile()* si occupa di creare il file compresso prendendo in input la radice dell'albero di Huffman e scendendo ricorsivamente i livelli di quest'ultimo fino ad arrivare al carattere che sto cercando, memorizzando ad ogni livello un 1 per ogni figlio destro da cui sono passato e uno 0 per ogni figlio sinistro. Quindi per ogni carattere del file in input scrivo in output la sua codifica.

- **Considerazioni finali:**

I risultati ottenuti sono i seguenti:

```
La lunghezza del file in input e' 100000 byte, mentre la lunghezza del file compresso e' 53543.75 byte
La percentuale di compressione e' 53.54375 %
```

La codifica ottenuta è istantanea e univocamente decifrabile e quindi avremo una compressione senza perdita di informazione. Inoltre, la codifica di Huffman e ottimale e quindi nessuna codifica istantanea per simboli può fornire una migliore compressione media.

Dal momento che l'entropia si avvicina moltissimo alla lunghezza attesa possiamo dire che è a ridondanza nulla e che non ci sono margini di miglioramento per la compressione ottenibile.