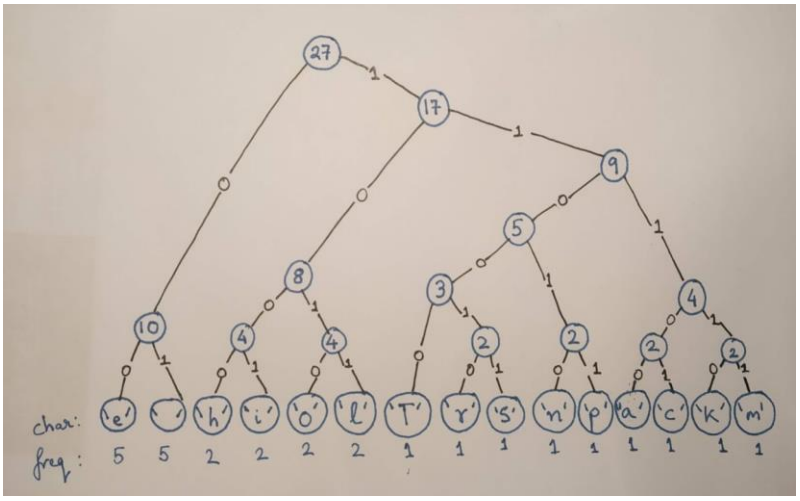
Assignment: Huffman Compression

Name: Rajit Banerjee
Student Number: 18202817

Huffman coding is a lossless data compression algorithm which assigns variable-length code to different characters in some input text.

Task 1: Drawing a Huffman tree for the phrase: "There is no place like home".



char x	frequency	P(x)	binary code	L(x)
'e'	5	5/27	00	2
' '	5	5/27	01	2
'h'	2	2/27	1000	4
'i'	2	2/27	1001	4
'o'	2	2/27	1010	4
'l'	2	2/27	1011	4
'T'	1	1/27	11000	5
'r'	1	1/27	110010	6
's'	1	1/27	110011	6
'n'	1	1/27	11010	5
'p'	1	1/27	11011	5
'a'	1	1/27	11100	5
'c'	1	1/27	11101	5
'k'	1	1/27	11110	5
'm'	1	1/27	11111	5

Characters (x) in the phrase, their frequencies and the resulting **Huffman codes** in binary (obtained by traversing the tree built above using the character-frequency values) are shown in the table (entries sorted by frequency and order of occurrence for convenience only).
P(x) is the probability of occurrence of character x in the phrase.
L(x) is the code length for character x.

Expected code length,

$$E(L) = \sum_i P(x_i) * L(x_i) = \left(\frac{5}{27}\right) * 2 * 2 + \left(\frac{2}{27}\right) * 4 * 4 + \left(\frac{1}{27}\right) * 7 * 5 + \left(\frac{1}{27}\right) * 2 * 6 = \mathbf{3.667} \text{ bits/char}$$

Shannon Entropy,

$$H(X) = \sum_i P(x_i) * \log_2 \frac{1}{P(x_i)} = \left(\frac{5}{27}\right) * 2.433 * 2 + \left(\frac{2}{27}\right) * 3.755 * 4 + \left(\frac{1}{27}\right) * 4.755 * 9 = \mathbf{3.599} \text{ bits/char}$$

Compression efficiency, $\eta = \frac{H(X)}{E(L)} = \frac{3.599}{3.667} = 98.14\%$

Compression ratio $= \frac{compressed\ bits}{original\ bits} = \frac{5*2+2*2+4*4+5*7+6*2}{27*8} = \frac{99}{216} = \mathbf{45.8}\%$

Therefore, the Huffman algorithm can reduce the size of the given phrase by **54.2%**, which is the same for every possible alternative Huffman encoding of the phrase with the same expected code length E(L). This is also the optimal prefix-free encoding for the phrase, as proved by David Huffman.

Task 2: Coding the Huffman algorithm in Java.

The class path in the repository is /src/a1/Huffman.java. In your terminal, change the directory to /src.
Compilation: javac a1/Huffman.java
Execution: java a1/Huffman compress inputFileName outputFileName
Execution: java a1/Huffman decompress inputFileName outputFileName
Dependencies: util/BinaryIn.java, util/BinaryOut.java, util/MinPQ.java, util/StdOut.java
Examples:
% java a1/Huffman compress a1/medTale.txt a1/medTale_comp.txt
% java a1/Huffman decompress a1/medTale_comp.txt a1/medTale_decomp.txt

Task 3: Compression Analysis

1. Experiments with compression:

Input File	Output File	Original Bits	Compressed Bits	Compression Ratio	Time (millisec)
genomeVirus.txt	genomeVirus_comp.txt	50008	12576	25.15%	12
medTale.txt	medTale_comp.txt	45872	24664	53.77%	14
mobydick.txt	mobydick_comp.txt	9708968	5505432	56.70%	127
q32x48.bin	q32x48_comp.bin	1536	816	53.13%	10
sample.txt	sampe_comp.txt	15416	9344	60.61%	12

```
\src>java a1/Huffman compress a1/genomeVirus.txt a1/genomeVirus_comp.txt

Time taken for compression:     12 milliseconds
Input file (original):          a1/genomeVirus.txt
Output file (compressed):       a1/genomeVirus_comp.txt
Original bits:                  50008
Compressed bits:                12576
Compression ratio:              12576/50008 = 25.15%
```

```
\src>java a1/Huffman compress a1/medTale.txt a1/medTale_comp.txt

Time taken for compression:     14 milliseconds
Input file (original):          a1/medTale.txt
Output file (compressed):       a1/medTale_comp.txt
Original bits:                  45872
Compressed bits:                24664
Compression ratio:              24664/45872 = 53.77%
```

```
\src>java a1/Huffman compress a1/mobydick.txt a1/mobydick_comp.txt

Time taken for compression:     127 milliseconds
Input file (original):          a1/mobydick.txt
Output file (compressed):       a1/mobydick_comp.txt
Original bits:                  9708968
Compressed bits:                5505432
Compression ratio:              5505432/9708968 = 56.70%
```

```
\src>java a1/Huffman compress a1/q32x48.bin a1/q32x48_comp.bin

Time taken for compression:     10 milliseconds
Input file (original):          a1/q32x48.bin
Output file (compressed):       a1/q32x48_comp.bin
Original bits:                  1536
Compressed bits:                816
Compression ratio:              816/1536 = 53.13%
```

```
\src>java a1/Huffman compress a1/sample.txt a1/sample_comp.txt

Time taken for compression:     12 milliseconds
Input file (original):          a1/sample.txt
Output file (compressed):       a1/sample_comp.txt
Original bits:                  15416
Compressed bits:                9344
Compression ratio:              9344/15416 = 60.61%
```

Analysis:

The running time of Huffman compression is N + R log R.

N is the input size and R is the alphabet size [Complexity: O(n log n)].

- genomeVirus.txt: This file contains genomic code (4 characters, ATCG with approximately equal frequencies). Therefore, the Huffman tree is balanced and each character is compressed from 8-bit ASCII to a 2-bit code. Hence, the compression ratio is about 25%.
- medTale.txt: This file contains English text from the medieval age. The compression ratio of 53.77% is pretty good since it reduces the file size by nearly half the original
- mobydick.txt: This is larger file that contains English text of nearly 10 million bits, and the Huffman algorithm takes over 120 milliseconds. However, the compression ratio of 56.7% is still good.
- q32x48.bin: This is a bitmap that the Huffman algorithm can compress with a ratio of 53.13%, thus reducing the size of the map by about half.
- sample.txt: Text containing commonly used English (history of the Huffman algorithm!). Compression ratio of 60.61% is close to that of mobydick.txt and medTale.txt (all English).

2. Experiments with decompression:

Input File	Output File	Final Bits	Time (millisec)
genomeVirus_comp.txt	genomeVirus_decomp.txt	50008	4
medTale_comp.txt	medTale_decomp.txt	45872	4
mobydick_comp.txt	mobydick_decomp.txt	9708968	57
q32x48_comp.bin	q32x48_decomp.bin	1536	1
sample_comp.txt	sampe_decomp.txt	15416	2

```
\src>java a1/Huffman decompress a1/genomeVirus_comp.txt a1/genomeVirus_decomp.txt

Time taken for decompression:   4 milliseconds
Input file (compressed):        a1/genomeVirus_comp.txt
Output file (decompressed):     a1/genomeVirus_decomp.txt
Final bits (decompressed):      50008
```

```
\src>java a1/Huffman decompress a1/medTale_comp.txt a1/medTale_decomp.txt

Time taken for decompression:   4 milliseconds
Input file (compressed):        a1/medTale_comp.txt
Output file (decompressed):     a1/medTale_decomp.txt
Final bits (decompressed):      45872
```

```
\src>java a1/Huffman decompress a1/mobydick_comp.txt a1/mobydick_decomp.txt

Time taken for decompression:   57 milliseconds
Input file (compressed):        a1/mobydick_comp.txt
Output file (decompressed):     a1/mobydick_decomp.txt
Final bits (decompressed):      9708968
```

```
\src>java a1/Huffman decompress a1/q32x48_comp.bin a1/q32x48_decomp.bin

Time taken for decompression:   1 milliseconds
Input file (compressed):        a1/q32x48_comp.bin
Output file (decompressed):     a1/q32x48_decomp.bin
Final bits (decompressed):      1536
```

```
\src>java a1/Huffman decompress a1/sample_comp.txt a1/sample_decomp.txt

Time taken for decompression:   2 milliseconds
Input file (compressed):        a1/sample_comp.txt
Output file (decompressed):     a1/sample_decomp.txt
Final bits (decompressed):      15416
```

Analysis:

The running time of Huffman decompression is much lower than the compression time because the trie has already been built, and the algorithm only needs to traverse the trie and expand the previously compressed bits. As seen in the decompression experiments above, the algorithm successfully recreates the original file from the compressed files, and the final bits obtained for the decompressed file are identical to the original uncompressed files.

The time taken to decompress the file simply depends on number of bits in the original uncompressed file. E.g. mobydick.txt (containing nearly 10 million uncompressed bits) takes 57 milliseconds to get decompressed, whereas q32x48.bin (containing only 1536 uncompressed bits) takes only 1 millisecond for decompression.

3. Compress already compressed files:

```
src>java a1/Huffman compress a1/sample_comp.txt a1/sample_comp2.txt

Time taken for compression:     16 milliseconds
Input file (original):          a1/sample_comp.txt
Output file (compressed):       a1/sample_comp2.txt
Original bits:                  9344
Compressed bits:                11640
Compression ratio:              11640/9344 = 124.57%

C:\Users\Rajit Banerjee\Desktop\Algorithms\algorithm-portfolio-20290-rajitbanerjee\
src>java a1/Huffman compress a1/q32x48_comp.bin a1/q32x48_comp2.bin

Time taken for compression:     12 milliseconds
Input file (original):          a1/q32x48_comp.bin
Output file (compressed):       a1/q32x48_comp2.bin
Original bits:                  816
Compressed bits:                1272
Compression ratio:              1272/816 = 155.88%
```

Compressed files sample_comp.txt and q32x48_comp.bin were compressed again using the Huffman algorithm. This resulted in the files increasing in size (compression ratio > 100%), as opposed to being compressed further. The reason is probably that the compression algorithm reads in data from the input file as 8-bit ASCII characters, but the data in the compressed file contains bits that don't represent any meaningful ASCII text. There is a large number of different symbols in the file, with varying frequencies. This increase in the number of different symbols increases the size of the Huffman tree, since more symbols occur with smaller frequencies. This increases the size of the file after the second compression.

4. RunLength vs Huffman algorithm:

```
src>java util/RunLength - < a1/q32x48.bin | java util/BinaryDump 0
1144 bits

C:\Users\Rajit Banerjee\Desktop\Algorithms\algorithm-portfolio-20290-rajitbanerjee\
src>java a1/Huffman compress a1/q32x48.bin a1/q32x48_comp.bin

Time taken for compression:     15 milliseconds
Input file (original):          a1/q32x48.bin
Output file (compressed):       a1/q32x48_comp.bin
Original bits:                  1536
Compressed bits:                816
Compression ratio:              816/1536 = 53.13%
```

The RunLength encoding compresses the bitmap from 1536 bits to 1144 bits.
Compression ratio = 1144/1536 = 74.48%

The Huffman algorithm compresses the same bitmap from 1536 bits to 816 bits.
Compression ratio = 816/1536 = 53.13%

Therefore, the Huffman algorithm provides approximately a 20% better compression ratio for the bitmap file, compared to RunLength encoding.
This is because the long runs of 0 bits and some 1 bits are all read in as 8-bit ASCII characters for the Huffman algorithm, thus providing fewer different symbols (with high frequencies). This reduces the number of nodes, the size of the Huffman trie, and consequently the size of the compressed file. In general, bitmaps are more suitable for Huffman compression compared to run length encoding, since the former only needs 8-bit characters to occur frequently in order to produce short codes, whereas the latter requires long runs of bits, i.e., it needs bits to occur consecutively as well as frequently, for efficiency.