Usage:

1) gcc huffman.c -lpthread
2) ./a.out <file_path> <num_threads>.

   Example:

./a.out enwik8.txt 4

Algorithm:

1) The file is divided into chunks of equal sizes and distributed among the threads.
2) The frequency count of all characters are calculated in parallel by each thread. Mutexes used to synchronize access to frequency array.
3) Huffman tree is built and the corresponding encoding of each character calculated sequentially with the help of a priority queue. The encodings and the huffman tree are available for each thread to access.
4) Then, the actual encoding is then performed parallelly again, using the character encodings calculated in the previous point. Each thread takes care of its respective chunk.
5) Then, the encoded file is decoded parallelly with the help of the huffman tree already built in step 4. Each thread takes care of its respective chunk.
6) The encoded file is saved as <file_name>-encoded and the decoded file as <file_name>-decoded in the same folder.

Q1)b)

All time measurements are in seconds and is performed over a randomly generated 20 Mb file.

NO. OF THREADS	SEQUENTIAL TIME	TOTAL TIME	RATIO
1	0.002152	2.597279	0.000829
2	0.001987	4.334538	0.000458
3	0.002198	4.925008	0.000446
4	0.001742	6.626954	0.000263
5	0.002593	9.057346	0.000286
6	0.001781	11.242948	0.000158
7	0.002144	14.899670	0.000144
8	0.002223	17.303080	0.000128

Q1)c)

Let p be the number of threads, and n be the size of the file. Time complexity of:

1) Character frequency calculation: O(n/p): which is the size of each thread's chunk.
2) Huffman tree building: O (n log n): There are n iterations (of concatenating any two sub-trees), and each iteration takes O (log n). (Time complexity of extracting minimum from a priority queue).
3) Generating the encoding of each character: O(n): Time complexity of traversing the whole tree. In the worst case, we may have a long chain of nodes, i.e a linked list.
4) Encoding of the file: O(n/p): which is the size of each thread's chunk.
5) Decoding of file: O(n/p): which is the size of each thread's chunk.

Total **asymptotic time complexity** will thus turn out to be O(n log n).

Speedup: O(n log n) / O (n log n):O(1). (This is independent of no. of processors)

Efficiency: Speedup / p: O (1/ p).

Cost: O (p * n/p) + O (n log n) = O (n log n). (As parallel time execution is O(n/p) as can be seen in (1), (4) and (5) above.

Iso-efficiency: As the efficiency is not dependent on the problem size but decreases with no. of processors, iso-efficiency is not defined in this case.

1)d)

In our experiments, the time taken is seen to increase with the no. of threads, as can be seen in the above table. So, the optimum no. of threads turn out to be 1 only. (The serial case).