Data compression :   
  
***Problem statement :***  
  
i. If you were to compress the messages sent between clients, which compression algorithm(s) would be most suited for this purpose? Justify your answer. Compress and decompress any one of the messages sent between any two nodes.   
  
ii. Compute the worst case time complexity of your solution in d(i).  
  
***Solution:***  
  
***Code :***  
  
import java.util.\*;

class HuffmanNode implements Comparable<HuffmanNode> {

int frequency;

char data;

HuffmanNode left, right;

public HuffmanNode(char data, int frequency) {

this.data = data;

this.frequency = frequency;

left = right = null;

}

@Override

public int compareTo(HuffmanNode o) {

return this.frequency - o.frequency;

}

}

public class Huffman {

public static void main(String[] args) {

String text = "Peter Kadhila is the true Chief, 222042192.";

Map<Character, Integer> frequencyMap = new HashMap<>();

for (char c : text.toCharArray()) {

frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);

}

PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<>();

for (Map.Entry<Character, Integer> entry : frequencyMap.entrySet()) {

priorityQueue.offer(new HuffmanNode(entry.getKey(), entry.getValue()));

}

while (priorityQueue.size() > 1) {

HuffmanNode left = priorityQueue.poll();

HuffmanNode right = priorityQueue.poll();

HuffmanNode parent = new HuffmanNode('$', left.frequency + right.frequency);

parent.left = left;

parent.right = right;

priorityQueue.offer(parent);

}

HuffmanNode root = priorityQueue.peek();

Map<Character, String> huffmanCodes = new HashMap<>();

generateCodes(root, "", huffmanCodes);

System.out.println("Huffman Codes:");

for (Map.Entry<Character, String> entry : huffmanCodes.entrySet()) {

System.out.println(entry.getKey() + " : " + entry.getValue());

}

}

public static void generateCodes(HuffmanNode root, String code, Map<Character, String> huffmanCodes) {

if (root == null) {

return;

}

if (root.left == null && root.right == null) {

huffmanCodes.put(root.data, code);

return;

}

generateCodes(root.left, code + "0", huffmanCodes);

generateCodes(root.right, code + "1", huffmanCodes);

}

}  
  
  
***Explanation and Answers:***  
Huffman coding is well-suited for compressing messages sent between clients for the following.

1. **Variable-Length Encoding**: Huffman coding provides variable-length encoding, meaning that different characters can be represented by codes of different lengths. Characters that occur more frequently in the input data are assigned shorter codes, while less frequent characters are assigned longer codes. This property helps in reducing the overall size of the encoded message, especially for messages with repetitive patterns or frequent occurrence of certain characters.
2. **Efficiency**: Huffman coding tends to produce efficient compression for messages with non-uniform character frequencies. In typical text-based communication, certain characters (like spaces, vowels, punctuation marks) occur more frequently than others. Huffman coding exploits these frequency variations to assign shorter codes to common characters, resulting in better compression ratios.
3. **No Redundancy**: Huffman coding generates prefix codes, which means that no code is a prefix of another code. This property ensures that there is no ambiguity in decoding the compressed message, simplifying the decompression process. Additionally, there is no redundant information in the encoded message, leading to optimal compression.
4. **Simple Implementation**: The algorithm for constructing Huffman trees and generating Huffman codes is relatively simple to implement. It involves building a binary tree based on character frequencies and assigning codes based on the tree structure. This simplicity makes Huffman coding efficient for real-time compression and decompression of messages sent between clients, even with limited computational resources.  
     
     
   Worst Case

The worst-case time complexity of the Huffman coding algorithm mainly depends on the number of unique characters in the input message and their frequencies. In the provided implementation:

* Constructing the frequency map for the input message takes O(n) time, where n is the length of the message.
* Building the priority queue of Huffman nodes also takes O(n) time, as each character is processed once.
* Constructing the Huffman tree by repeatedly dequeuing and enqueuing nodes from the priority queue takes O(n log n) time, where n is the number of unique characters.
* Generating Huffman codes by traversing the Huffman tree takes O(n) time, where n is the number of unique characters.

The worst-case time complexity of the Huffman coding algorithm implemented in this code is O(n log n), where n is the number of unique characters in the input message. This complexity makes Huffman coding suitable for real-time compression and decompression of messages sent between clients, even for large datasets.