Name : Sneha Singh
ID : 231080167
Branch - SY BTech It

---

1. Theory Section:

Huffman Coding Algorithm:

Principle of Huffman Coding:

Huffman coding is a **lossless data compression algorithm** that minimizes the average length of the code for each character based on their frequencies. The primary idea is to assign shorter codes to more frequent characters and longer codes to less frequent characters. This reduces the total number of bits needed to represent the data, achieving compression.

Building the Huffman Tree:

1. **Character Frequencies**: The process starts by calculating the frequency of each character in the given data.
2. **Priority Queue**: A min-heap or priority queue is used to build the tree. The nodes in the heap represent characters and their frequencies.
3. **Tree Construction**:
    - Two nodes with the smallest frequencies are extracted from the heap.
    - A new node is created, which is the parent of these two nodes. The frequency of this new node is the sum of the frequencies of its children.
    - The new node is inserted back into the heap.
    - This process is repeated until there is only one node left in the heap, which becomes the **root of the Huffman Tree**.
4. **Assigning Codes**: Once the tree is built, binary codes are assigned by traversing from the root to each leaf node (representing a character). Moving left assigns a `0`, and moving right assigns a 1. Each character gets a unique binary code based on its position in the tree.

Role of Binary Trees:

The binary tree structure in Huffman coding allows us to efficiently assign binary codes to characters. By following the left or right child at each level of the tree, we construct a binary sequence (code). The Huffman Tree ensures that:

- Characters with higher frequencies are placed closer to the root (shorter codes).

- Characters with lower frequencies are placed further from the root (longer codes).

Prefix-Free Codes:

Huffman coding uses **prefix-free codes**, which means that no code is a prefix of another. This ensures that the encoded data can be uniquely decoded without ambiguity. For example, if we have two characters encoded as `101` and `1010`, the first code could be misinterpreted during decoding, so prefix-free encoding eliminates this issue.

Time and Space Complexity:

- **Time Complexity**:
  - Building the priority queue (min-heap) takes O(nlogn), where n is the number of unique characters.
  - Constructing the tree involves removing and inserting nodes in the heap, which takes O(nlogn)
  - Hence, the overall time complexity is **O(nlogn)**.
- **Space Complexity**:
  - The space complexity depends on storing the frequency table, the heap, and the tree.
  - Storing the frequency table requires O(n) space.
  - The heap requires O(n) space, and the Huffman Tree itself also uses O(n)space.
  - Hence, the overall space complexity is **O(n)**.

Limitations of Huffman Coding:

1. **Not Suitable for Small Data**: For very small datasets, the overhead of storing the Huffman Tree may outweigh the benefits of compression.
2. **Equal Frequency Characters**: When characters have nearly equal frequencies, the compression might not be very efficient, as the generated codes will have similar lengths.
3. **Static vs. Adaptive Huffman**: Standard Huffman coding is static; it requires a full pass over the data to build the frequency table before compression begins. In scenarios where the data is streamed or too large to fit in memory, adaptive Huffman coding may be needed, but it is more complex.
4. **No Context Sensitivity**: Huffman coding works on individual characters and does not account for the context in which characters appear. Algorithms like Lempel-Ziv-Welch (LZW) or Arithmetic Coding might achieve better compression for data with repetitive patterns or contextual dependencies.
5. **Fixed Alphabet**: Huffman coding is limited to a fixed set of characters (alphabet). It is not efficient for data where the set of symbols is dynamic or where characters appear sporadically.