1.Dynamic programming (DP) is a technique for solving problems by breaking them into smaller subproblems, solving each subproblem once, and storing the results to avoid redundant work. It's useful for optimization problems where the goal is to find the best solution efficiently.

2. A greedy algorithm is a problem-solving technique that makes the locally optimal choice at each step with the hope of finding the global optimum. In other words, it always picks the best option available at the moment, without considering the bigger picture or future consequences.

3. Greedy Algorithm makes a local optimal choice at each step, hoping it leads to the global optimum whereas Dynamic Programming solves the problem by breaking it down into subproblems, solving each subproblem optimally and combining their solutions.

4. In the Greedy Approach, we begin at node 2, where we have two choices: 7 and 5. Since 7 is greater than 5, we move to 7. From node 7, we can go to either 3 or 6, and since 6 is greater than 3, we choose 6. Therefore, the greedy path is 2 → 7 → 6, and the sum is 2 + 7 + 6 = 15.

For the Dynamic Programming approach, we start by calculating the maximum sum for the leaf nodes. For 3 and 6, the maximum sum is simply their value (3 and 6, respectively). Similarly, for 13 and 11, the maximum sum is 13 and 11. Next, we calculate the maximum sum for node 7, which leads to 3 and 6. The sum at 7 is 7 + max(3, 6) = 13. For node 5, which leads to 13 and 11, the sum is 5 + max(13, 11) = 18. Finally, at the root node 2, which leads to 7 and 5, the maximum sum is 2 + max(13, 18) = 20. Thus, the optimal path is 2 → 5 → 13, with a sum of 20.

5. We first need to analyze the frequency of each character in the string. The characters and their frequencies are as follows: **z** appears 4 times, **b** appears 6 times, **u** appears 3 times, **e** appears 3 times, and the **space** character appears 2 times. Next, we build the Huffman tree by starting with the least frequent characters and combining them until we have a single tree. The characters with the lowest frequencies (space and u) are combined first, followed by combining e with the previous node, and so on, until we merge all nodes into a final tree. Based on this tree, the Huffman codes for each character are assigned: **b** gets the shortest code (1 bit), **z** gets 2 bits, **e** gets 3 bits, and both **space** and **u** get 4 bits. Using these Huffman codes, we calculate the total compressed size by multiplying the frequency of each character by the length of its Huffman code. The compressed size comes out to 43 bits. In contrast, the original size of the text using ASCII encoding is calculated by multiplying the number of characters (18) by 8 bits per character, giving a total of 144 bits. Finally, the compression ratio is computed by dividing the compressed size (43 bits) by the original size (144 bits), resulting in a compression ratio of approximately **0.298**. This means the compressed text takes up about 29.8% of the original size, reflecting a significant reduction in space due to Huffman coding.