LAB 8

SE172806 – Phạm Hoàng Minh Khôi

1)

To decode the given string, we can match each binary code with its corresponding letter based on the provided code:

A: 0 B: 100 C: 1010 D: 1011 E: 1100 F: 1101 G: 1110 H: 1111

Decoding the string "100010100101101100011010100100000111001111" using the above mappings, we get:

1 000 1010 0101 1011 0001 1010 1001 0000 0111 0011 11

Matching each part with the corresponding letter, we have:

1 = B 000 = A 1010 = C 0101 = E 1011 = D 0001 = A 1010 = C 1001 = F 0000 = A 0111 = G 0011 = D 11 = H

Therefore, the decoded string "100010100101101100011010100100000111001111" translates to the letters "BACEADCFAGDH".

2)

**A.** Using Huffman encoding, the code for character F can be determined by constructing a Huffman tree based on the character frequencies:

100% (Root)

/ \

B 68%

/ \

C 40%

/ \

D 24%

/ \

G 16%

/

E 6%

/

F 10%

Following the path from the root to character F, we get the code "10".

**B.** Encoding the text "AABCDEFEEFG" using the Huffman codes:

A: 100

B: 0

C: 110

D: 1110

E: 1111

F: 10

G: 1100

The encoded text becomes "100011001011101111011010".

To calculate the average code length, we sum the lengths of the encoded strings for each character and divide by the total number of characters:

(3 + 1 + 3 + 4 + 4 + 2 + 4) / 11 = 3.0909

So, the average code length of the encoded string is approximately 3.0909.

**C.** To check whether the string "10010101011011001101010111" is a valid encoded string of the above codes, we need to traverse the codes and verify if they correspond to valid characters.

Starting from the beginning of the string:

"1" matches character "B".

"00" does not match any valid code.

"1" matches character "B".

"01" does not match any valid code.

"0" matches character "A".

"101" does not match any valid code.

"010" does not match any valid code.

"1" matches character "B".

"010" does not match any valid code.

"011" does not match any valid code.

"0" matches character "A".

"101" does not match any valid code.

"01" does not match any valid code.

"01" does not match any valid code.

"011" does not match any valid code.

"010" does not match any valid code.

"111" does not match any valid code.

In conclusion, the string "10010101011011001101010111" is not a valid encoded string of the given codes because it contains sequences of bits that do not correspond to any valid character code.

**3)**

In a Huffman encoding of an alphabet with n characters, the maximal length of a codeword is n - 1.

This is because the Huffman algorithm constructs a binary tree where each character is represented by a path from the root to a leaf node. The codeword for each character is determined by the binary digits along the path, where each left branch corresponds to a 0 and each right branch corresponds to a 1.

In the worst case scenario, the Huffman tree forms a completely unbalanced tree, where one character has a codeword of length n - 1 (as it is the deepest leaf node), and all other characters have codewords of shorter length.

Therefore, the maximal length of a codeword in a Huffman encoding of an alphabet with n characters is n - 1.

**4)**

If the alphabet's characters are given in a sorted order of their frequencies, it is indeed possible to construct a Huffman tree in linear time.

The key idea is to use a min-heap (priority queue) data structure to store the frequencies and characters of the alphabet. Initially, each character is treated as a single-node tree with its frequency as the key. Then, the two trees with the lowest frequencies are repeatedly merged until only one tree remains, which becomes the Huffman tree.

Here is the step-by-step process to construct the Huffman tree in linear time:

1. Create a min-heap and insert each character with its corresponding frequency.

2. While there is more than one tree in the min-heap:

- Extract the two trees with the lowest frequencies from the min-heap.

- Create a new tree with these two trees as children, and the frequency as the sum of their frequencies.

- Insert the new tree back into the min-heap.

3. The remaining tree in the min-heap is the Huffman tree.

Since the characters are already sorted by their frequencies, the extraction of the two lowest frequency trees can be done efficiently using a min-heap. The insertion of the new tree into the min-heap also takes constant time. Overall, the number of operations is proportional to the number of characters in the alphabet, resulting in linear time complexity.

Therefore, when the characters of the alphabet are given in a sorted order of their frequencies, a Huffman tree can be constructed in linear time.

**5)**

If we apply the run-length encoding algorithm to the raw message "BBBBBUUUUXXXUUPPPPPPPUUKKKKKKKK", the compressed output will represent the consecutive occurrences of each character in the message as a count followed by the character itself.

Here's the compressed output:

5B4U3X2UP7UK8K

Explanation:

- The initial "B" appears 5 times consecutively, so we represent it as "5B".

- The subsequent "U" appears 4 times consecutively, so we represent it as "4U".

- The next "X" appears 3 times consecutively, so we represent it as "3X".

- The following "U" appears again, but this time only once, so we represent it as "1U".

- The letter "P" appears 7 times consecutively, so we represent it as "7P".

- The next "U" appears once, so we represent it as "1U".

- Finally, the letter "K" appears 8 times consecutively, so we represent it as "8K".

Combining all the compressed parts, we get the compressed output: "5B4U3X2UP7UK8K".

**6)**

Using the LZW algorithm to encode the message "ABBABABACCDABCCDB" with the initial dictionary containing the entries: (1) A, (2) B, (3) C, (4) D, we can go through the encoding process step by step.

**A.** The strings contained in the dictionary after the encoding process are:

(1) A

(2) B

(3) C

(4) D

(5) AB

(6) BA

(7) AC

(8) CD

(9) DAB

(10) BC

(11) CC

(12) ABCC

(13) CDB

**B.** The compressed string is the sequence of indices corresponding to the entries in the dictionary.

The encoding process steps:

1. Start with the first character "A".

2. Move to the next character "B".

3. Check if the current string "AB" exists in the dictionary. Since it does, move to the next character "B".

4. Check if the current string "ABB" exists in the dictionary. Since it does not, add it to the dictionary as entry (14).

5. Output the index of the previous string "AB" (index 5) and continue to the next character "A".

6. Check if the current string "BA" exists in the dictionary. Since it does not, add it to the dictionary as entry (15).

7. Output the index of the previous string "B" (index 2) and continue to the next character "B".

8. Check if the current string "AB" exists in the dictionary. Since it does, move to the next character "A".

9. Check if the current string "ABA" exists in the dictionary. Since it does not, add it to the dictionary as entry (16).

10. Output the index of the previous string "AB" (index 5) and continue to the next character "B".

11. Check if the current string "BA" exists in the dictionary. Since it does, move to the next character "B".

12. Check if the current string "BAB" exists in the dictionary. Since it does not, add it to the dictionary as entry (17).

13. Output the index of the previous string "BA" (index 6) and continue to the next character "A".14. Check if the current string "AB" exists in the dictionary. Since it does, move to the next character "C".

15. Check if the current string "ABC" exists in the dictionary. Since it does not, add it to the dictionary as entry (18).

16. Output the index of the previous string "AB" (index 5) and continue to the next character "C".

17. Check if the current string "BC" exists in the dictionary. Since it does not, add it to the dictionary as entry (19).

18. Output the index of the previous string "C" (index 3) and continue to the next character "C".

19. Check if the current string "CC" exists in the dictionary. Since it does, move to the next character "D".

20. Check if the current string "CD" exists in the dictionary. Since it does not, add it to the dictionary as entry (20).

21. Output the index of the previous string "CC" (index 11) and continue to the next character "A".

22. Check if the current string "AB" exists in the dictionary. Since it does, move to the next character "B".

23. Check if the current string "ABD" exists in the dictionary. Since it does not, add it to the dictionary as entry (21).

24. Output the index of the previous string "AB" (index 5) and continue to the next character "C".

25. Check if the current string "BC" exists in the dictionary. Since it does, move to the next character "C".

26. Check if the current string "BCC" exists in the dictionary. Since it does not, add it to the dictionary as entry (22).

27. Output the index of the previous string "BC" (index 10) and continue to the next character "C".

28. Check if the current string "CC" exists in the dictionary. Since it does, move to the next character "D".

29. Check if the current string "CD" exists in the dictionary. Since it does, move to the next character "B".

30. Check if the current string "CDB" exists in the dictionary. Since it does not, add it to the dictionary as entry (23).

31. Output the index of the previous string "CD" (index 8) and continue to the next character "C".

32. Check if the current string "CC" exists in the dictionary. Since it does, move to the next character "D".

33. Check if the current string "CD" exists in the dictionary. Since it does, move to the next character "B".

34. Check if the current string "CDB" exists in the dictionary. Since it does, move to the next character "C".

35. Check if the current string "DBC" exists in the dictionary. Since it does not, add it to the dictionary as entry (24).

The compressed string is the sequence of indices:

5 2 5 6 5 3 11 5 10 11 8 10 23 11 8 3 24

Therefore, the compressed string is "5 2 5 6 5 3 11 5 10 11 8 10 23 11 8 3 24".