

Session 4

Huffman's Coding Algorithm

Huffman Coding is a popular **greedy algorithm** used for δ . It is particularly effective for sequences where certain characters appear more frequently than others.

Note

Compression is a technique used to reduce the overall size of data, making storage and transmission more efficient.

How the Algorithm Works

The core idea behind Huffman Coding is to assign variable-length binary codes to characters based on their frequency:

- **Frequency Analysis:** The algorithm counts how many times each character occurs in the dataset.
- **Tree Construction:** It builds a binary tree (the **Huffman Tree**) from the bottom up, starting with the least frequent characters.
- **Binary Encoding:** In this tree, more frequent characters are placed closer to the root, resulting in shorter binary strings. Conversely, rarer characters receive longer codes.

Key Benefits

- **Significant Reduction:** Huffman algorithms can typically compress data size by 70–80%, depending on the redundancy of the source material.
- **Optimal Prefix Codes:** It ensures that no code is a prefix of another, which prevents ambiguity during the decompression (decoding) process.
- *Efficiency:* By utilizing a greedy approach, the algorithm consistently finds the most mathematically optimal way to represent characters as binary strings for a given frequency set.

Objective

✗ Do not copy. Read for understanding and the viva

The main objectives of this session are to:

- Implement Huffman's Coding algorithm.
- Test the implementation on different problem instances
- Construct the optimal binary prefix code

Question 1

Problem Statement

■ Write in lab record

Apply Huffman's algorithm to construct an optimal binary prefix code for the letters and its frequencies in the following table.

Letters	A	B	I	M	S	X	Z
Frequency	10	7	15	8	10	5	2

Show the complete steps.

Explanation

✗ Do not copy. Read for understanding and the viva

Using your data: A(10), B(7), I(15), M(8), S(10), X(5), Z(2).

- **Initialize:** Create a leaf node for each character.
- **Merge (Greedy Step):** Pick the two smallest: Z(2) and X(5).
 - **Create** a parent node with frequency $2 + 5 = 7$.
- **Repeat:** Now pick the next two smallest from the remaining list. This continues until only one node (the **Root**) remains.
- **Assign Bits:** Starting from the Root, assign 0 to every left branch and 1 to every right branch.

Implementation

■ Write in lab record

■

Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

huffman.py

```
1 import heapq
2
3 class Node:
4     def __init__(self, char, freq):
5         self.char = char
6         self.freq = freq
7         self.left = None
8         self.right = None
9
10    def __lt__(self, other):
11        return self.freq < other.freq
12
13    def solve_huffman():
14        # 1. Initialize data from the problem table
15        data = {'A': 10, 'B': 7, 'I': 15, 'M': 8, 'S': 10, 'X': 5, 'Z': 2}
16        heap = [Node(c, f) for c, f in data.items()]
17        heapq.heapify(heap)
18
19        print("--- STEP-BY-STEP MERGING (GREEDY STEPS) ---")
20        step = 1
21        while len(heap) > 1:
22            # Extract two smallest frequencies
23            n1 = heapq.heappop(heap)
24            n2 = heapq.heappop(heap)
25
26            # Merge them
27            merged = Node(None, n1.freq + n2.freq)
28            merged.left = n1
29            merged.right = n2
30
31            print(f"Step {step}: Merged {n1.char or 'Node'}({n1.freq}) and "
32                  f"{n2.char or 'Node'}({n2.freq}) → New Node({merged.freq})")
33
34            heapq.heappush(heap, merged)
35            step += 1
36
37        # 2. Generate Codes and Print Table
38        root = heap[0]
39        codes = {}
40
41        def generate_codes(node, current_code):
42            if node:
43                if node.char:
```

```

44         codes[node.char] = current_code
45         generate_codes(node.left, current_code + "0")
46         generate_codes(node.right, current_code + "1")
47
48     generate_codes(root, "")
49
50     print("\n--- FINAL OPTIMAL BINARY PREFIX CODES ---")
51     print(f"{'Letter':<10} | {'Frequency':<10} | {'Huffman Code':<15}")
52     print("-" * 40)
53     for char in sorted(data.keys()):
54         print(f"{char:<10} | {data[char]:<10} | {codes[char]:<15}")
55
56 solve_huffman()

```

Sample Output

Write in lab record

```

1  --- STEP-BY-STEP MERGING (GREEDY STEPS) ---
2  Step 1: Merged Z(2) and X(5) → New Node(7)
3  Step 2: Merged B(7) and Node(7) → New Node(14)
4  Step 3: Merged M(8) and A(10) → New Node(18)
5  Step 4: Merged S(10) and Node(14) → New Node(24)
6  Step 5: Merged I(15) and Node(18) → New Node(33)
7  Step 6: Merged Node(24) and Node(33) → New Node(57)
8
9  --- FINAL OPTIMAL BINARY PREFIX CODES ---
10 Letter      | Frequency  | Huffman Code
11 -----
12 A           | 10        | 111
13 B           | 7         | 010
14 I           | 15        | 10
15 M           | 8         | 110
16 S           | 10        | 00
17 X           | 5         | 0111
18 Z           | 2         | 0110

```

SH

Question 2

Problem Statement

■ Write in lab record

Implement Huffman's coding algorithm and run it on the problem instance of Q1.

Implementation

■ Write in lab record

Implementation

Refer to Question 1, Implementation. It's the proper solution for this question.

Question 3

Problem Statement

■ Write in lab record

What is an optimal binary tree and Huffman code for the following set of frequencies? Find out the average number of bits required per character. Show complete steps.

A:15 B:25 C:5 D:7 E:10 F:13 G:9

Initial Frequencies - Sorted

■ Write in lab record

Char	C	D	G	E	F	A	B
Freq	5	7	9	10	13	15	25

Total Frequency =

Huffman Tree

Write in lab record

Step	Nodes in Pool (Sorted)	Merged Nodes	New Node
1	C:5, D:7, G:9, E:10, F:13, A:15, B:25	C(5) + D(7)	N1:12
2	N1:12, G:9, E:10, F:13, A:15, B:25 → Sort: G:9, E:10, N1:12...	G(9) + E(10)	N2:19
3	N1:12, F:13, A:15, N2:19, B:25	N1(12)+F(13)	N3:25
4	A:15, N2:19, N3:25, B:25	A(15)+N2(19)	N4:34
5	N3:25, B:25, N4:34	N3(25)+B(25)	N5:50
6	N4:34, N5:50	N4(34)+N5(50)	Root:84

Assigning Codes

Write in lab record

Convention: 0 = Left Child, 1 = Right Child

1 Root(84)

2 | 0 → N4(34)

3 | | 0 → A(15) → 00

4 | | 1 → N2(19)

5 | | | 0 → G(9) → 010

6 | | | 1 → E(10) → 011

7 | 1 → N5(50)

8 | | 0 → N3(25)

9 | | | 0 → N1(12)

10 | | | | 0 → C(5) → 1000

11 | | | | 1 → D(7) → 1001

12 | | | 1 → F(13) → 101

13 | 1 → B(25) → 11

SH

Huffman Codes

■ Write in lab record

Char	Frequency	Code	Length
A	15	00	2
B	25	11	2
C	5	1000	4
D	7	1001	4
E	10	011	3
F	13	101	3
G	9	010	3

Bits/Character

× Do not copy. Read for understanding and the viva

```
1 Total Bits = Σ (Frequency × Code Length)
2             = (15×2) + (25×2) + (5×4) + (7×4) + (10×3) + (13×3) + (9×3)
3             = 30 + 50 + 20 + 28 + 30 + 39 + 27 = 224
4
5 Average Bits = Total Bits / Total Frequency
6             = 224 / 84 ≈ 2.67 bits/character
```

SH

Implementations

■ Write in lab record

■

Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

huffman.py

```
1 import heapq
2
3 class Node:
4     def __init__(self, freq, char=None):
5         self.freq = freq
6         self.char = char
7         self.left = None
8         self.right = None
9
10    # Required for heapq to compare nodes by frequency
11    def __lt__(self, other):
12        return self.freq < other.freq
13
14    def build_huffman_tree(freq_dict):
15        """Builds Huffman Tree using a Min-Heap"""
16        heap = [Node(f, c) for c, f in freq_dict.items()]
17        heapq.heapify(heap)
18
19        while len(heap) > 1:
20            left = heapq.heappop(heap)
21            right = heapq.heappop(heap)
22            merged = Node(left.freq + right.freq)
23            merged.left = left
24            merged.right = right
25            heapq.heappush(heap, merged)
26
27        return heap[0] if heap else None
28
29    def generate_codes(root, current_code="", codes=None):
30        """DFS traversal to assign 0/1 codes"""
31        if codes is None:
32            codes = {}
33        if root is None:
34            return
35        if root.char is not None: # Leaf node
36            codes[root.char] = current_code
37            return
38        generate_codes(root.left, current_code + "0", codes)
39        generate_codes(root.right, current_code + "1", codes)
40        return codes
41
42    def main():
43        freq = {'A': 15, 'B': 25, 'C': 5, 'D': 7, 'E': 10, 'F': 13, 'G': 9}
```

```
44     root = build_huffman_tree(freq)
45     codes = generate_codes(root)
46
47     # Calculate average bits
48     total_bits = sum(freq[c] * len(codes[c]) for c in codes)
49     total_freq = sum(freq.values())
50     avg_bits = total_bits / total_freq
51
52     print("Huffman Codes:")
53     for char in sorted(codes.keys()):
54         print(f"{char}: {codes[char]}")
55     print(f"\nAverage bits per character: {avg_bits:.2f}")
56
57 if __name__ == "__main__":
58     main()
```

Sample Output

■ Write in lab record

```
1 Huffman Codes:
2 A: 00
3 B: 11
4 C: 1000
5 D: 1001
6 E: 011
7 F: 101
8 G: 010
9
10 Average bits per character: 2.67
```

SH

Question 4

Problem Statement

■ Write in lab record

A file contains only colon, spaces, new line character and digits in the following frequency:
colon(80), space(500), new line(110), commas(500), 0(300), 1(200), 2(150), 3(60), 4(180),

5(240), 6(170), 7(200), 8(202).

Using the Huffman's algorithm to construct an optimal binary prefix code.

Frequencies - Sorted

Char	Frequency
3	60
colon	80
new line	110
2	150
6	170
4	180
1	200
7	200
8	202
5	240
0	300
space	500
comma	500

Total Frequency = 2892

Merge Steps

- We are using min-heap approach
- `\n` - New line

Step	Two Smallest Nodes	Merged Node (Freq)	Pool After Merge (Sorted)
1	3(60) + colon(80)	N1:140	\n:110, N1:140, 2:150, 6:170, 4:180, 1:200, 7:200, 8:202, 5:240, 0:300, space:500, comma:500
2	\n(110) + N1(140)	N2:250	2:150, 6:170, 4:180, 1:200, 7:200, 8:202, 5:240, N2:250, 0:300, space:500, comma:500
3	2(150) + 6(170)	N3:320	4:180, 1:200, 7:200, 8:202, 5:240, N2:250, 0:300, N3:320, space:500, comma:500
4	4(180) + 1(200)	N4:380	7:200, 8:202, 5:240, N2:250, 0:300, N3:320, N4:380, space:500, comma:500
5	7(200) + 8(202)	N5:402	5:240, N2:250, 0:300, N3:320, N4:380, N5:402, space:500, comma:500
6	5(240) + N2(250)	N6:490	0:300, N3:320, N4:380, N5:402, N6:490, space:500, comma:500
7	0(300) + N3(320)	N7:620	N4:380, N5:402, N6:490, N7:620, space:500, comma:500
8	N4(380) + N5(402)	N8:782	N6:490, N7:620, N8:782, space:500, comma:500
9	N6(490) + space(500)	N9:990	N7:620, N8:782, N9:990, comma:500
10	comma(500) + N7(620)	N10:1120	N8:782, N9:990, N10:1120
11	N8(782) + N9(990)	N11:1772	N10:1120, N11:1772
12	N10(1120) + N11(1772)	Root:2892	Tree Complete

Tree

SH

```

1 Root(2892)
2   └─ 0 → N10(1120)
3     └─ 0 → comma(500)      → 00
4     └─ 1 → N7(620)
5         └─ 0 → 0(300)      → 010
6         └─ 1 → N3(320)
7             └─ 0 → 2(150)   → 0110
8             └─ 1 → 6(170)   → 0111
9   └─ 1 → N11(1772)
10      └─ 0 → N8(782)
11          └─ 0 → N4(380)
12              └─ 0 → 4(180)   → 1000
13              └─ 1 → 1(200)   → 1001
14              └─ 1 → N5(402)
15                  └─ 0 → 7(200)   → 1010
16                  └─ 1 → 8(202)   → 1011
17      └─ 1 → N9(990)
18          └─ 0 → N6(490)
19              └─ 0 → 5(240)   → 1100
20              └─ 1 → N2(250)
21                  └─ 0 → \n(110) → 11010
22                  └─ 1 → N1(140)
23                      └─ 0 → 3(60)   → 110110
24                      └─ 1 → colon(80) → 110111
25      └─ 1 → space(500)      → 111

```

Huffman Codes

■ Write in lab record

Char	Freq	Code	Length
space	500	00	2
0	300	010	3
2	150	0110	4
6	170	0111	4
4	180	1000	4
1	200	1001	4
7	200	1010	4
8	202	1011	4
5	240	1100	4
\n	110	11010	5
3	60	110110	6
:	80	110111	6
,	500	111	3

Bits/Character

× Do not copy. Read for understanding and the viva

1 **Total Bits** = $\Sigma(\text{Frequency} \times \text{Code Length})$ SH
 2 $\Rightarrow (500 \times 2) + (300 \times 3) + (150 \times 4) + (170 \times 4) + (180 \times 4) + (200 \times 4) + (200 \times 4) + (202 \times 4) + (240 \times 4) + (110 \times 5) + (60 \times 6) + (80 \times 6)$
 3 $\Rightarrow 1000 + 900 + 600 + 680 + 720 + 800 + 800 + 808 + 960 + 550 + 360 + 480 +$
 4 $\Rightarrow 10,158 \text{ bits}$
 5
 6 **Average Bits** = $\text{Total Bits} / \text{Total Frequency}$
 7 $\Rightarrow 10,158 / 2,892 \approx 3.51 \text{ bits/character}$

Implementations

■ Write in lab record

■

Lab record: write one language only. Pick yours once and every page opens on it; the other tabs are the same solution for comparison.

Python

huffman.py

```
1 import heapq
2
3
4 class Node:
5     """Huffman Tree Node"""
6
7     def __init__(self, freq, char=None):
8         self.freq = freq
9         self.char = char # None for internal nodes
10        self.left = None
11        self.right = None
12
13    def __lt__(self, other):
14        """Enable comparison for heapq (min-heap by frequency)"""
15        return self.freq < other.freq
16
17
18 def build_huffman_tree(freq_dict):
19     """Build Huffman tree using priority queue (min-heap)"""
20     heap = [Node(f, c) for c, f in freq_dict.items()]
21     heapq.heapify(heap)
22
23     while len(heap) > 1:
24         # Extract two smallest frequency nodes
25         left = heapq.heappop(heap)
26         right = heapq.heappop(heap)
27
28         # Create merged internal node
29         merged = Node(left.freq + right.freq)
30         merged.left = left
31         merged.right = right
32         heapq.heappush(heap, merged)
33
34     return heap[0] if heap else None
35
36
37 def generate_codes(node, current_code="", codes=None):
38     """DFS traversal to assign binary codes (0=left, 1=right)"""
39     if codes is None:
40         codes = {}
41     if node is None:
42         return codes
43     if node.char is not None: # Leaf node
```

```
44     codes[node.char] = current_code
45     return codes
46     generate_codes(node.left, current_code + "0", codes)
47     generate_codes(node.right, current_code + "1", codes)
48     return codes
49
50
51 def calculate_avg_bits(codes, freq_dict):
52     """Calculate weighted average bits per character"""
53     total_bits = sum(freq_dict[c] * len(codes[c]) for c in codes)
54     total_freq = sum(freq_dict.values())
55     return total_bits / total_freq
56
57
58 def main():
59     # Input frequencies
60     freq = {
61         ":": 80,
62         " ": 500,
63         "\n": 110,
64         ",": 500,
65         "0": 300,
66         "1": 200,
67         "2": 150,
68         "3": 60,
69         "4": 180,
70         "5": 240,
71         "6": 170,
72         "7": 200,
73         "8": 202,
74     }
75
76     # Build tree and generate codes
77     root = build_huffman_tree(freq)
78     codes = generate_codes(root)
79
80     # Display results
81     print("Huffman Codes:")
82     for char in sorted(codes.keys(), key=lambda c: freq[c], reverse=True):
83         display = repr(char) if char in [" ", "\n", ":", ","] else char
84         print(f"{display:>6} (freq={freq[char]:4d}): {codes[char]}")
85
86     avg = calculate_avg_bits(codes, freq)
87     print(f"\nAverage bits per character: {avg:.2f}")
```

```
88     print(f"Total frequency: {sum(freq.values())}")
89     print(f"Total bits for encoded file: {sum(freq[c] * len(codes[c]) for c
90
91
92 if __name__ == "__main__":
93     main()
```

Sample Output

■ Write in lab record

```
1 Huffman Codes:
2 ' ' (freq= 500): 00
3 ', ' (freq= 500): 111
4 0 (freq= 300): 010
5 5 (freq= 240): 1100
6 8 (freq= 202): 1011
7 1 (freq= 200): 1001
8 7 (freq= 200): 1010
9 4 (freq= 180): 1000
10 6 (freq= 170): 0111
11 2 (freq= 150): 0110
12 '\n' (freq= 110): 11010
13 ':' (freq= 80): 110111
14 3 (freq= 60): 110110
15
16
17 Average bits per character: 3.512448132780083
18 Total frequency: 2892
19 Total bits for encoded file: 10,158
```

SH

✎ Edit this page

< Previous
Session 3

Next >
Session 5