AZA - Practical Assignment Analysis

Leon Radó

December 4, 2025

1 Scheduling with Deadlines

1.1 Complexity Analysis

```
sort(jobs.begin(), jobs.end(), compareProfit);
```

This line sorts all n jobs by profit. The C++ sorting algorithm time complexity is:

$$O(n \cdot \log n)$$

```
for (i = 1; i < n; i++) {
    K = J;
    K.push_back(jobs[i]);

    if (isFeasible(K)) {
        J = K;
    }
}
```

This loop executes n times. Copying the current set J into K takes $O(i)$ time in iteration i. The feasibility check called inside the loop takes $O(i \cdot \log i)$:

$$\sum_{i=1}^{n} (i + i \cdot \log i)$$

```
sort(K.begin(), K.end(), compareDeadline);
```

Inside the feasibility check, this sort operates on a list of size $|K| = i$:

$$O(i \cdot \log i)$$

```
for (int i = 0; i < K.size(); i++) {
    int timeUnit = i + 1;

    if (timeUnit > K[i].deadline) {
        return false;
    }
}
```

After sorting, the feasibility check iterates through all $|K| = i$ jobs to verify deadlines:

$$O(i)$$

—

Total Time Complexity:[1]

$$\boxed{T(n) = \sum_{i=1}^{n} (i + i \cdot \log i) = O(n^2 \cdot \log n)}$$

—

Total Space Complexity:

- List of jobs: $O(n)$
- Temporary list K: $O(n)$
- Scheduled list J: $O(n)$

$$\boxed{S(n) = O(n)}$$

[1] The time required to print the final solution is not included in the complexity analysis.

1.2 Complexity Analysis (Disjoint Set)

```
sort(jobs.begin(), jobs.end(), compareProfit);
```

This line sorts all n jobs by profit. The C++ sorting algorithm time complexity is:

$$O(n \cdot \log n)$$

```
int d = 0;
for (auto& job : jobs) {
    d = max(d, job.deadline);
}
```

Iterates over each job to find the maximum deadline:

$$O(n)$$

```
for (int i = 0; i <= d; i++) {
    makeSet(i);
}
```

Initialization runs $d+1$ times, which is simplified to d:

$$O(d)$$

```
for (auto& job : jobs) {
    int root = find(job.deadline);
    int slot = small(root);

    if (slot == 0) continue;

    S[slot - 1] = job;
    merge(find(slot), find(slot - 1));
}
```

This loop processes each job exactly once:

- **find** searches for the root of the group containing the deadline, and in worst-case takes $O(\log m)$

- **small** is simple access and runs in constant time $O(1)$
- **merge** is also approximated to constant time $O(1)$

$$O(n \cdot \log m)$$

$m = \min(n, d)$ represents the number of slots actually used.

—

Total Time Complexity:[2]

$$\boxed{T(n) = n \cdot \log n + n + d + n \cdot \log m = O(n \cdot \log n)}$$

—

Total Space Complexity:

- List of jobs: $O(n)$
- Disjoint Set structure U: $O(d+1)$
- Scheduled list S: $O(d)$

$$\boxed{S(n) = O(n)}$$

2 Huffman Compression

2.1 Solution Proposal

The compressed file produced by the implementation is divided into 4 logical parts: **a header with code lengths, the code definitions, the encoded data, and a padding byte**.

1. The header starts with exactly 256 bytes. Each of these bytes represents the length of the code for a specific character value from 0 to 255. A length of zero means that the character does not appear in the input and therefore does not have a code assigned to it.
2. Afterward there are written generated codes for each present character bit by bit. This section contains the raw bits of all codes in order from character 0 to character 255. Because the decompressing mechanism already knows the lengths from the header, it can read exactly the correct number of bits for each character.

[2]The time required to print the final solution is not included in the complexity analysis.

3. Following the code definitions, the compressing mechanism writes the actual compressed data. Each byte of the original input is replaced by its Huffman code. This produces continuous stream of bits representing the entire original file. Bits are packed into bytes without any gaps in between.

4. Because the total number of bits in the compressed file may not be a multiple of eight, the compressing mechanism writes one extra byte at the very end of the file. This final byte contains the number of unused padding bits in the last data byte. A padding value of zero means the last byte is fully used, while a positive value indicates how many of the least significant bits must be ignored during decompression.

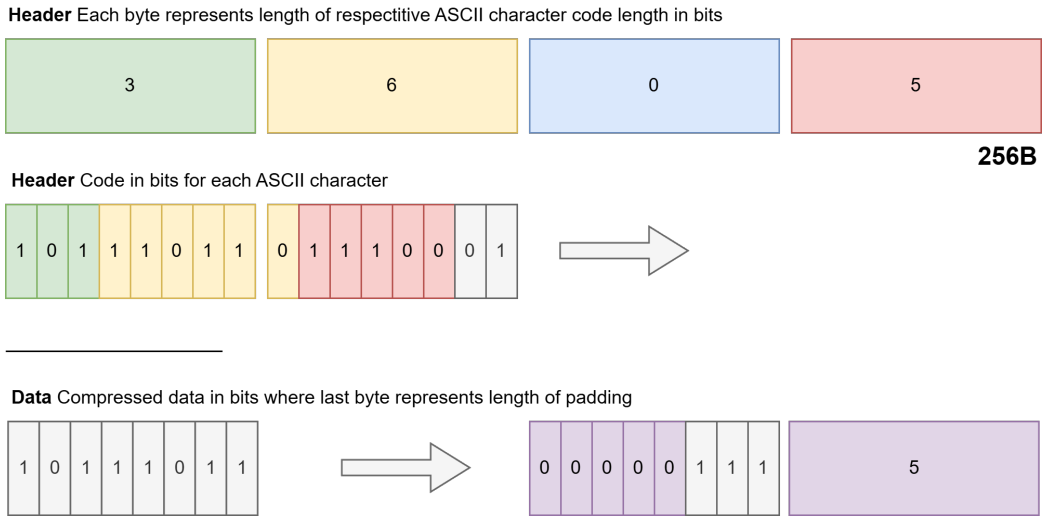

Figure 1: Compressed File Scheme

2.2 Complexity Analysis (Compression)

```
1  ifstream input("input.txt", ios::binary);
2  vector<unsigned char> data(istreambuf_iterator<char>(input), {});
3  input.close();
```

Reading file and copying its content into vector:

$$O(n)$$

```
1  vector<unsigned int> frequencies(256, 0);
2
```

```
for (auto byte : data) {
    frequencies[byte]++;
}
```

Counting frequencies of present character per each byte:

$$O(n)$$

```
for (int i = 0; i < 256; i++) {
    if (frequencies[i] > 0) {
        minHeap.push(make_shared<HuffmanNode>(i, frequencies[i]));
    }
}
```

Inserting character nodes into min heap:

$$O(a \cdot \log a)$$

a = number of distinct characters from the input.

```
while (minHeap.size() > 1) {
    auto left = minHeap.top();
    minHeap.pop();

    auto right = minHeap.top();
    minHeap.pop();

    auto parent = make_shared<HuffmanNode>(
        '\0', left->freq + right->freq
    );

    parent->left = left;
    parent->right = right;

    minHeap.push(parent);
}

root = minHeap.top();
```

This loop executes $a-1$ times, each iteration performs two pop operations and one push operation on the min heap, each costing $O(\log a)$:

$$(a-1) \cdot \log a = O(a \cdot \log a)$$

```
if (node->isLeaf()) {
    codes[node->data] = code.empty() ? "0" : code;
    return;
}

generateCodes(node->left, code + "0");
generateCodes(node->right, code + "1");
```

Traversal of the Huffman tree, each node (internal $a-1$ and leaf a is visited exactly once. Operations per node are all $O(1)$:

$$2a-1 = O(a)$$

```
for (int i = 0; i < 256; i++) {
    char c = (char)i;
    uint8_t len = coder.codes.count(c) ? coder.codes[c].size() : 0;

    outBytes.push_back(len);
}
```

Writing header of the compressed file, containing the lengths of all codes:

$$O(256) = O(1)$$

```
for (int i = 0; i < 256; i++) {
    char c = (char)i;

    if (coder.codes.count(c)) {
        for (char bit : coder.codes[c]) {
            writeBit(bit == '1');
        }
    }
}
```

Writing code definitions for each character. Each bit of the code is written using **writeBit**, which is $O(1)$:

$$O(c) = O(a)$$

c = maximum length of code definition, in worst case can be approximated to a.

```
for (unsigned char c : data) {
    for (char bit : coder.codes[c]) {
        writeBit(bit == '1');
```

```
4         }
5     }
```

Writing encoded data, replaced with code definitions:

$$O(n \cdot a)$$

```
1 output.write((char*)outBytes.data(), outBytes.size());
```

Writing buffer with the header to the output file:

$$O(n \cdot a)$$

—

Total Time Complexity:[3]

$$\boxed{\begin{aligned} T(n,a) &= n + n + a \cdot \log a + a \cdot \log a + a + a + n \cdot a + n \cdot a \\ &= O(n \cdot a) \end{aligned}}$$

$$\boxed{T(n) = O(n)}$$

—

Total Space Complexity:

- Input data vector: $O(n)$
- Frequency table: $O(1)$
- Min heap for Huffman tree: $O(a)$
- Huffman tree itself: $O(a)$
- Map of codes: $O(a^2)$
- Output buffer: $O(n \cdot a)$

$$\boxed{S(n,a) = O(n \cdot a)}$$

$$\boxed{S(n) = O(n)}$$

[3] Second formula applies if we assume that we are working with ASCII characters and consider a as constant

2.3 Complexity Analysis (Decompression)

```
ifstream input("output.bin", ios::binary);
vector<uint8_t> bytes(istreambuf_iterator<char>(input), {});
input.close();
```

Reading compressed file into vector:

$$O(n \cdot c) = O(n \cdot a)$$

c = maximum length of code definition, in worst case can be approximated to a.

```
vector<uint8_t> lengths(256);

for (int i = 0; i < 256; i++) {
    lengths[i] = bytes[i];
}
```

Reading the header of 256 bytes with code lengths for each character:

$$O(256) = O(1)$$

```
for (int i = 0; i < 256; i++) {
    ...
    for (int j = 0; j < len; j++) {
        bitset<8> bits(bytes[bytePos]);
        int b = bits[7 - bitPos];
        code.push_back(b ? '1' : '0');
        ...
    }
    codes[code] = (char)i;
}
```

Reconstructing Huffman codes for all present characters:

$$O(a \cdot c) = O(a^2)$$

```
for (bytePos; bytePos < end; bytePos++) {
    bitset<8> bits(bytes[bytePos]);
    for (int b = 7 - bitPos; b >= 0; b--) {
        currCode.push_back(bits[b] ? '1' : '0');
        if (codes.count(currCode)) {
```

```
6                 output.put(codes[currCode]);
7                 currCode.clear();
8             }
9         }
10        bitPos = 0;
11    }
```

Decoding the main data stream, each bit is processed once, with code lengths shorter or equal to c:

$$O(n \cdot c) = O(n \cdot a)$$

```
1  ofstream output("decoded.txt", ios::binary);
2  ...
3  output.close();
```

Writing buffer with decompressed data to the output file:

$$O(n)$$

—

Total Time Complexity:[4]

$$\boxed{T(n,a) = n \cdot a + a^2 + n \cdot a + n = O(n \cdot a)}$$

$$\boxed{T(n) = O(n)}$$

—

Total Space Complexity:

- Input data vector: $O(n \cdot a)$
- Code lengths array: $O(1)$
- Map of codes: $O(a^2)$
- Current code string: $O(a)$
- Output buffer: $O(n)$

$$\boxed{S(n,a) = O(n \cdot a)}$$

$$\boxed{S(n) = O(n)}$$

[4]Second formula applies if we assume that we are working with ASCII characters and consider a as constant

2.4 Algorithm Improvements

- **Bitwise operations:** Instead of writing bits individually in function calls as in the current solution, use bitwise operations and write them afterward.
- **Frequency table for present characters:** Instead of storing frequencies for all 256 characters, only work with characters that are present in the input.
- **Canonical Huffman codes:** Use canonical Huffman codes so it's not needed to store whole code definitions in the header.