Aufgabe 1: Schmucknachrichten

Teilnahme-ID: 74130

Bearbeiter/-in dieser Aufgabe:
Matthew Greiner

April 21, 2025

Contents

1 Teilaufgabe A

1.1 Lösungsidee

Verständnis der Aufgabe

Die Grundidee dieser Aufgabe ist es einen gegebenen Unicode-Text mit Hilfe von Perlen zu kodieren, die alle denselben Durchmesser haben. Die Anzahl der Farben der Perlen ist gegeben und beträgt mindestens zwei. Das Ziel ist es, den gegebenen Text so mit den Perlen zu kodieren, dass eine möglichst kurze Perlenkette daraus gemacht werden kann. Die verwendete Codetabelle für die präfixfreie Kodierung und die Gesamtlänge der Botschaft muss am Ende ausgegeben werden.

Überlegungen zum Lösungsansatz

Da der Durchmesser aller Perlen gleich ist, muss die Anzahl der verwendeten Perlen in der Kodierung minimiert werden, um die Perlenkette möglichst kurz zu halten. Da jedes Zeichen aus dem Eingabetext einzeln durch einen Code aus Perlen dargestellt werden muss, macht es Sinn, dass häufig verwendete Zeichen kurze Codes und selten verwendete Zeichen längere Codes bekommen. Diese Anforderungen eine **optimale präfixfreie Kodierung basierend auf Zeichenhäufigkeiten zur Minimierung der Gesamtlänge** ist eine klassische Anwendung der Huffman-Kodierung, die auch als Basis für die Lösung dieser Teilaufgabe verwendet wird.

Die Huffman-Kodierung ist ein Algorithmus, der unter folgenden Bedingungen eine optimale binäre Kodierung liefert:

1. Jedes Zeichen bekommt einen eindeutigen Binärcode
2. Die Nachricht ist verlustfrei und eindeutig dekodierbar
3. Die durchschnittliche Länge (gewichtet nach Häufigkeit) wird minimiert

Da diese Bedingungen genau zu dieser Aufgabe passen, eignet sich diese Kodierung für diese Aufgabe gut. Standardmäßig funktioniert die Huffman-Kodierung aber binär. Da wir die gegebene Nachricht evtl. aber mit mehr als zwei Farben von Perlen kodieren müssen, muss der Algorithmus auf eine **n-ären** Huffman-Kodierung erweitert werden. Eine solche Verallgemeinerung des Algorithmus wurde bereits im Originalpaper von Huffman [1] beschrieben.
Diese Erweiterung des Algorithmus funktioniert analog zu der binären Huffman-Kodierung, nur die Erstellung des Huffman-Baums muss angepasst werden. Bei der normalen Huffman-Kodierung werden zunächst die zu kodierenden Zeichen in Knoten gespeichert. Dann werden die **zwei** Knoten mit den geringsten Häufigkeiten kombiniert und als Kinder von einem neuen Knoten gespeichert. Die "Häufigkeit" des neuen Knotens wird als Summe der beiden Häufigkeiten der Kinder festgelegt. Dieser Prozess wird wiederholt, bis nur noch ein Knoten übrig ist. Dieser ist dann die Wurzel des neuen Baums. Das bedeutet, bei jeder Kombinierung, verringert sich die Anzahl der betrachteten Knoten um eins. Der Baum wird also "bottom-up" erstellt, und garantiert somit eine optimale Kodierung (im Gegensatz zur Shannon-Fano-Kodierung, die einen Baum "top-down" erstellt und nicht immer die optimale Kodierung findet).

Der Unterschied zu der **n-nären** Huffman-Kodierung ist nur, dass die n Knoten mit den geringsten Häufigkeiten kombiniert werden und nicht nur zwei. Das hat zur Folge, dass mit jeder Kombination die Anzahl der betrachteten Knoten nun um $n-1$ sinkt. Daher muss gezielt darauf geachtet werden, dass nach allen Kombinationen nur noch ein Knoten übrigbleibt. Der Aufbau des n-nären Baums funktioniert, wenn folgende Gleichung erfüllt ist:

$$N \bmod (n-1) = 1 \quad (N = \text{Anzahl der Knoten}) \tag{1}$$

Wenn das nicht der Fall ist, können Platzhalterknoten hinzugefügt werden, die die Häufigkeit 0 haben, bis diese Bedingung erfüllt ist. Bei der Erstellung der Codetabelle wird der Graph (analog zur Standard Huffman-Kodierung) z.B. mit Tiefensuche traversiert. Dabei wird jeder Kante von einem Elternknoten zu einem Kindknoten eindeutig eine Zahl (z.B. von 0 bis k-1) zugeordnet. Eine Zahl repräsentiert in dieser Aufgabe eine Perlenfarbe. In der binären Version werden nur die Zahlen 0 und 1 benutzt.

1.2 Umsetzung

Um diesen Algorithmus umzusetzen und somit die Aufgabe zu lösen, wird wie folgt vorgegangen (die Nummerierung entspricht auch der Nummerierung in den Quellcode):

1. **Einlesen des Eingabetextes und der Anzahl der verschiedenen Perlenfarben**
   Da alle Eingaben in Dateien vorliegen, welche nur 3 Zeilen haben, werden sie zunächst eingelesen und an den Zeilenumbrüchen getrennt. Der Text sowie die Anzahl der verschiedenen Perlenfarben werden anschließend in entsprechenden Variablen gespeichert.

2. **Analyse der Häufigkeit der Zeichen im Text**
   Um zu zählen, wie häufig die Zeichen in dem Text vorkommen, wird über alle Zeichen des Texts iteriert. Die Werte werden in einer HashMap gespeichert, wobei das Zeichen den Schlüssel und die Anzahl seines Vorkommens im Text den zugehörigen Wert darstellt.

3. **Erstellung einen n-nären Huffman-Baums basierend auf den Häufigkeiten**
   Um den Baum zu erstellen, wird eine Klasse Node definiert, die einen Knoten im Huffman-Baum repräsentiert. Diese Klasse speichert das Zeichen, seine Häufigkeit, eine Liste seiner Kindknoten und einen Marker, ob der Knoten ein Blatt im Baum ist. Die einzelnen Knoten werden in einer PriorityQueue verwaltet, welche in Java als Min-Heap implementiert ist. Dadurch stehen die Knoten mit der geringsten Häufigkeit immer oben in der Warteschlange und diese Datenstruktur eignet sich daher gut, da für die Erstellung des Baums jeweils die n Knoten mit den **geringsten** Häufigkeiten kombiniert werden.

   Ablauf
   Zunächst werden die Zeichen des Text in Blattknoten gespeichert und in die PriorityQueue eingefügt.

Um aus diesen Knoten den Huffman-Baum zu konstruieren, kann man Platzhalter-Knoten einführen (siehe 1.1), allerdings gibt es auch einen Weg diese zusätzlichen Knoten zu vermeiden:
Statt Platzhalter mit der Häufigkeit 0 zu erzeugen, werden beim ersten Merging nicht zwingend n Knoten kombiniert, sondern eine bestimmte Anzahl r. Dieses r muss so gewählt werden, dass nach der ersten Kombination eine Gesamtanzahl an Knoten entsteht, mit der sich der Rest des Baums in gleichmäßigen n-ner Gruppen weiterbauen lässt.

Es gilt die Gleichung 1.
Wenn eine n-närer Baum gefordert ist, ist **N** die Anzahl an Blattknoten in der Queue. Die Gleichung muss auch nach dem ersten Merge mit r Knoten gelten:

$$(N - r + 1) \bmod (n - 1) = 1$$

Nach r umstellen ergibt:

$$\begin{aligned} N - r &\equiv 0 \mod (n-1) \\ \Rightarrow \quad r &= N \bmod (n-1) \end{aligned}$$

r muss mindestens 2 sein, damit ein Merge Sinn macht, daher wird:

$$r = (N - 2) \bmod (n - 1) + 2 \tag{2}$$

in der Implementierung verwendet.
In der Implementierung wird r also einmalig für den ersten Merge verwendet. Danach werden Gruppen zu je n Knoten kombiniert, bis nur noch ein Knoten übrig ist, das dann die Wurzel des Huffman-Baums ist.

4. **Erzeugen der Codetabelle anhand des Baums**
   Zur Speicherung der Codetabelle wird eine Hashmap verwendet, in der die Zeichen als Schlüssel hat und ihre zugehörige n-äre Huffman-Kodierung als Werte abgelegt sind. Die Werte werden dann schrittweise bei der Traversierung des Baums schrittweise aufgebaut.
   Um die Kodierung für jedes Zeichen zu finden, wird der Baum mit einer rekursiven Tiefensuche (DFS) durchlaufen. Dabei wird bei jedem rekursiven Aufruf der bisher zurückgelegte Pfad vom Wurzelknoten bis zum Aktuellen Knoten als String übergeben.

   Weil es sich um eine n-äre Huffman-Kodierung handelt, werden die Kanten des Baums durch die Ziffern 0 bis $n - 1$ beschrieben. Bei jedem Abstieg zu einem Kindknoten wird an die aktuelle Zeichenkette die Position des jeweiligen Kindes im Elternknoten angehängt. Wenn ein Blattknoten erreicht wird, also ein Knoten, der tatsächlich ein Zeichen repräsentiert und keine weiteren Nachfolger besitzt, wird der aktuell aufgebaute Pfad als Kodierung für dieses Zeichen in der Codetabelle gespeichert.

5. **Berechnung der Gesamtlänge der kodierten Nachricht**
   Für die Berechnung der Gesamtlänge der kodierten Nachricht wird die Häufigkeit jedes Zeichens (bereits in Schritt 2 bestimmt) mit der Länge seines entsprechenden Huffman-Codes multipliziert. Die Gesamtlänge ergibt sich aus der Summe dieser Produkte multipliziert mit dem Durchmesser einer Perle.

6. **Ausgabe der Codetabelle und der Gesamtlänge**
   Die Tabelle wird nach Code-Länge und Frequenz sortiert (optional, allerdings ein "nice to have" um direkt ablesen zu können, welche Zeichen am meisten vorkommen, wenn dies auch ausgegeben wird, was in dieser Implementierung gemacht wird). Anschließend wird sie formatiert ausgegeben sowie die Gesamtlänge der kodierten Nachricht aus 5.

1.3 Komplexitätsanalyse

Nun wird die Komplexität der einzelnen Schritte des Algorithmus betrachtet. Dabei ist:

- L die Länge des Eingabetextes,
- k die Anzahl der verschiedenen Zeichen (Symbole),

- n die Basis des Huffman-Baums (Anzahl erlaubter Kindknoten je Knoten).

Schritt	**Laufzeit**	**Speicherbedarf**
Einlesen der Datei	$\mathcal{O}(L)$	$\mathcal{O}(L)$
Häufigkeitsanalyse	$\mathcal{O}(L)$	$\mathcal{O}(k)$
Aufbau des Huffman-Baums	$\mathcal{O}(k \log k)$	$\mathcal{O}(k)$
Generieren der Codetabelle	$\mathcal{O}(k)$	$\mathcal{O}(k)$
Berechnung der Gesamtlänge	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Ausgabe Codetabelle (mit Sortierung)	$\mathcal{O}(k \log k)$	$\mathcal{O}(k)$
Gesamt	$\boldsymbol{\mathcal{O}(L + k \log k)}$	$\boldsymbol{\mathcal{O}(L + k)}$
Dominante Terme	*O(L) potentiell langer Text* *O(k log k) wegen Heap Operation/Sortierung*	

Table 1: Laufzeit- und Speicherkomplexität Teilaufgabe A

- **Einlesen und Häufigkeitsanalyse:** Beide Vorgänge durchlaufen den Text vollständig und brauchen daher eine Laufzeit von $\mathcal{O}(L)$.
- **Baumkonstruktion:** Wegen der Prioritätswarteschlange (Min-Heap) mit k Blattknoten entstehen $\frac{k-1}{n-1}$ Merges. Jede Insert- und Delete Operation im Heap braucht $\mathcal{O}(\log k)$, was insgesamt zu $\mathcal{O}(k \log k)$ führt.
- **Codetabelle und Gesamtlänge:** Beide Schritte durchlaufen jeden Knoten genau einmal und sind daher in $\mathcal{O}(k)$.
- **Ausgabe:** Die Codetabelle wird nach Frequenz und Code-Länge sortiert, was im Worst Case ebenfalls $\mathcal{O}(k \log k)$ erfordert. (Ohne die optionale Sortierung wäre das das in $\mathcal{O}(k)$)

Insgesamt ist die Laufzeit dominiert durch das Einlesen des Textes und die Sortiervorgänge beim Baumaufbau mit dem Min-Heap. Der Speicherbedarf ist moderat und wächst linear mit der Eingabegröße sowie der Anzahl unterschiedlicher Zeichen.
Daher erweist sich dieser Algorithmus auch bei größeren Texten als leistungsfähig.

1.4 Beispiele

Hier die Ausgaben des Programmes zu den Beispielen auf der BwInf-Webseite:

1. **schmuck0.txt**

```
Codetabelle:
(Jede Ziffer steht für eine Perlenfarbe z.B. könnte '0' rot bedeuten und '1' blau)
{
  '␣': 111 (Freq: 5)
  'E': 110 (Freq: 5)
  'I': 100 (Freq: 4)
  'N': 101 (Freq: 4)
  'S': 000 (Freq: 3)
  'R': 0100 (Freq: 2)
  'D': 0010 (Freq: 2)
  'L': 0111 (Freq: 2)
  'M': 0110 (Freq: 2)
  'O': 0011 (Freq: 2)
  'C': 01010 (Freq: 1)
  'H': 01011 (Freq: 1)
}

Gesamtlänge der Botschaft 113 (Anzahl in Perlen) bzw. 11.3cm
```

2. **schmuck00.txt**

```
Codetabelle:
(Jede Ziffer steht für eine Perlenfarbe z.B. könnte '0' rot bedeuten und '1' blau)
{
  '␣': 21 (Freq: 24)
  'e': 12 (Freq: 18)
  'i': 11 (Freq: 16)
  't': 02 (Freq: 10)
  'n': 01 (Freq: 9)
  'h': 220 (Freq: 8)
  's': 222 (Freq: 8)
  'c': 202 (Freq: 7)
  'l': 201 (Freq: 6)
  'a': 100 (Freq: 4)
  'r': 102 (Freq: 4)
  'd': 001 (Freq: 3)
  'D': 2212 (Freq: 3)
  'u': 2211 (Freq: 3)
  'G': 1012 (Freq: 2)
  'g': 2210 (Freq: 2)
  'm': 2000 (Freq: 2)
  'w': 2002 (Freq: 2)
  'E': 0001 (Freq: 1)
  'P': 0020 (Freq: 1)
  'W': 0002 (Freq: 1)
  'Z': 1010 (Freq: 1)
  'b': 0022 (Freq: 1)
  'f': 1011 (Freq: 1)
  'k': 0021 (Freq: 1)
  'o': 0000 (Freq: 1)
  'A': 20010 (Freq: 1)
  '?': 20011 (Freq: 1)
}

Gesamtlänge der Botschaft 372 (Anzahl in Perlen) bzw. 37.2cm
```

3. **schmuck01.txt**

```
Codetabelle:
(Jede Ziffer steht für eine Perlenfarbe z.B. könnte '0' rot bedeuten und '1' blau)
{
  '␣': 2 (Freq: 85)
  'e': 0 (Freq: 66)
  'n': 43 (Freq: 53)
  'r': 41 (Freq: 34)
  'i': 40 (Freq: 33)
  's': 33 (Freq: 26)
  'a': 31 (Freq: 22)
  'l': 32 (Freq: 22)
  't': 30 (Freq: 22)
  'h': 13 (Freq: 17)
  'c': 12 (Freq: 15)
  'g': 11 (Freq: 14)
  'o': 10 (Freq: 14)
  'm': 443 (Freq: 13)
  'u': 442 (Freq: 12)
  '.': 441 (Freq: 10)
  'd': 424 (Freq: 9)
```

```
    'k': 440 (Freq: 9)
    'E': 423 (Freq: 8)
    ',': 422 (Freq: 8)
    'D': 344 (Freq: 7)
    'b': 421 (Freq: 7)
    'w': 343 (Freq: 6)
    'ü': 341 (Freq: 5)
    'I': 340 (Freq: 4)
    'S': 144 (Freq: 4)
    'f': 143 (Freq: 4)
    'p': 142 (Freq: 4)
    'v': 141 (Freq: 4)
    'V': 140 (Freq: 3)
    'F': 4442 (Freq: 3)
    'O': 4443 (Freq: 3)
    'ö': 4441 (Freq: 3)
    'z': 4444 (Freq: 3)
    'B': 4440 (Freq: 2)
    'G': 4203 (Freq: 2)
    'ä': 4204 (Freq: 2)
    'H': 3420 (Freq: 1)
    'K': 3421 (Freq: 1)
    'M': 3422 (Freq: 1)
    'N': 3424 (Freq: 1)
    'R': 4200 (Freq: 1)
    'W': 4202 (Freq: 1)
    'ß': 3423 (Freq: 1)
    '...': 4201 (Freq: 1)
}

Gesamtlänge der Botschaft 1150 (Anzahl in Perlen) bzw. 115.0cm
```

1.5 Quellcode

Wichtige Teile des Quellcodes:

1. Einlesung der Daten

```java
// 1. Read input
private static InputWrapper parseInput(String input) {
    // Save every line in array of Strings
    String[] lines = input.split("\n");
    List<Integer> list = new ArrayList<>();
    for (String s : lines[1].split("␣")) {
        list.add(Integer.valueOf(s));
    }
    return new InputWrapper(Integer.parseInt(lines[0].trim()), lines[2], list);
}
```

2. Häufigkeitsanalyse

```java
// 2. Calculate character frequency from input text
private static Map<Character, Long> buildFrequencyMap(String text) {
    Map<Character, Long> frequencyMap = new HashMap<>();
    for (char character : text.toCharArray()) {
        frequencyMap.put(character, frequencyMap.getOrDefault(character, 0L) + 1);
    }
    return frequencyMap;
}
```

3. Erstellung einen n-nären Huffman-Baums

```java
// 3. Build n-ary Huffman tree and returns root Node
private static Node buildHuffmanTree(Map<Character, Long> frequencyMap, int n) {
    // PriorityQueue for nodes (sorted by frequency, lower frequency first)
    PriorityQueue<Node> priorityQueue = new PriorityQueue<>();

    // Create leaf nodes (nodes with characters) and add to priority queue
    for (Map.Entry<Character, Long> entry : frequencyMap.entrySet()) {
        priorityQueue.add(new Node(entry.getKey(), entry.getValue()));
    }

    int numSymbols = priorityQueue.size(); // Number of unique symbols (leaf nodes)

    // Determine how many nodes should be merged in the first step
    int r;
    if (numSymbols <= 1) { // Handle edge case: if 0 or 1 symbol, no merging needed
        r = numSymbols;
    } else {
        // Formula: find smallest valid r so that:
        // After merging r nodes -> remaining nodes + 1 new node
        // -> total node count allows full n-ary merges
        int remainder = (numSymbols - 2) % (n - 1);
        r = remainder + 2; // Ensures 2 <= r <= n
    }

    // Build Huffman tree
    while (priorityQueue.size() > 1) {
        // Use r only for first merge if needed
        int nodesToMerge = (priorityQueue.size() == numSymbols) ? r : n;
        // Cannot merge more nodes than available
        nodesToMerge = Math.min(nodesToMerge, priorityQueue.size());

        if (nodesToMerge < 2) {
            // Should not happen if n > 1
            System.out.println("irgendwas ist broken");
            break;
        }

        List<Node> children = new ArrayList<>();
        long mergedFrequency = 0;

        // Extract nodes (num of nodesToMerge) with lowest frequencies
        for (int i = 0; i < nodesToMerge; i++) {
            Node node = priorityQueue.poll();
            children.add(node);
            mergedFrequency += node.frequency;
        }

        // Create new internal node with new values
        Node internalNode = new Node(children, mergedFrequency);

        // Add new internal node back to priority queue
        priorityQueue.add(internalNode);
    }

    // Last node in queue is root of Huffman tree
    return priorityQueue.poll();
}
```

4. Erzeugen der Codetabelle

```java
// 4. Generate Huffman codes by traversing tree with recursion and DFS
private static void generateCodes(Node node, String currentCode, Map<Character, String>
    codeTable, int numColors) {
    // Base case
    if (node == null) {
        return;
    }

    // If it is a leaf, it represents a character, then assign a code
    if (node.isLeaf()) {
```

```
        if (node.character != null) {
            codeTable.put(node.character, currentCode);
        }
        // Leaf is reached, stop recursion
        return;
    }

    // Assign codes 0 until k-1 to children
    for (int i = 0; i < node.children.size(); i++) {
        generateCodes(node.children.get(i), currentCode + i, codeTable, numColors);
    }
}
```

5. Berechnung der Gesamtlänge

```
// 5. Calculate total length of encoded message
private static long calculateTotalLength(Map<Character, String> codeTable,
    Map<Character, Long> frequencyMap,
        int diameter) {
    long totalLength = 0;
    // Iterate over each Character
    for (Map.Entry<Character, Long> entry : frequencyMap.entrySet()) {
        totalLength += entry.getValue() * codeTable.get(entry.getKey()).length();
    }
    return totalLength * diameter;
}
```

6. Ausgabe

```
// Print code table in a readable format
private static void printCodeTable(Map<Character, String> codeTable, Map<Character,
    Long> frequencyMap) {
    if (codeTable.isEmpty()) {
        System.out.println("{}");
        return;
    }
    List<Character> sortedKeys = new ArrayList<>(codeTable.keySet());
    // Sorting is optional
    // Sort by length of code
    sortedKeys.sort((a, b) -> Integer.compare(codeTable.get(a).length(),
    codeTable.get(b).length()));
    // Sort by frequency
    sortedKeys.sort((a, b) -> Long.compare(frequencyMap.get(b), frequencyMap.get(a)));

    System.out.println("{");
    for (Character character : sortedKeys) {
        // Handle special characters for printing
        String c;
        c = switch (character) {
            case '␣' -> "␣";
            case '"' -> c = "\\\"";
            default -> String.valueOf(character);
        };
        System.out.println("␣␣'" + c + "':␣" + codeTable.get(character) + "␣(Freq:␣"
                + frequencyMap.get(character) + ")");
    }
    System.out.println("}");
}
```

2 Teilaufgabe B

2.1 Lösungsidee

2.2 Umsetzung

2.3 Komplexitätsanalyse

2.4 Beispiele

2.5 Quellcode

References

[1] David A. Huffman (1952) *A Method for the Construction of Minimum-Redundancy Codes*, *Proceedings of the IRE*, 40(9), 1098–1101. https://doi.org/10.1109/JRPROC.1952.273898