Aufgabe 1: Schmucknachrichten

Teilnahme-ID: 74130

Bearbeiter/-in dieser Aufgabe:
Matthew Greiner

April 23, 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.
   Dies ist was den Huffman Algorithmus zu einem *greedy* Algorithmus macht. Er trifft in jedem Schritt eine lokal optimale Entscheidung (kombiniere die n unwahrscheinlichsten Knoten), die nachweislich zur globalen optimalen Lösung führt.

   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. Die Ziffern entsprechen allgemein formuliert, den Kodierungsbuchstaben, die in diesem Fall die jeweilige Perlenart bzw. Perlenfarbe darstellt. 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 (in dieser Dokumentation auch als Quellsymbole bezeichnet),
- 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:** Dieser Schritt ist durchläuft jeden Knoten (k Knoten) genau einmal mit DFS (Tiefensuche). Pro Knoten wird eine Schleife über alle n Kinder ausgeführt. Bei der n-nären Kodierung hat der Baum im schlimmsten Fall $\mathcal{O}(k)$ Knoten. Da n eine kleine Konstante ist, beträgt die Laufzeit für diesen Schritt $\mathcal{O}(k)$.
- **Gesamtlänge:** Dieser Schritt iteriert über alle verschiedenen Symbole, die einen Code bekommen und ist 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 und garantiert eine optimale Kodierung.

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;
```

```
40
            // Extract nodes (num of nodesToMerge) with lowest frequencies
42          for (int i = 0; i < nodesToMerge; i++) {
                Node node = priorityQueue.poll();
44              children.add(node);
                mergedFrequency += node.frequency;
46          }

48          // Create new internal node with new values
            Node internalNode = new Node(children, mergedFrequency);
50
            // Add new internal node back to priority queue
52          priorityQueue.add(internalNode);
        }
54
        // Last node in queue is root of Huffman tree
56      return priorityQueue.poll();
    }
```

4. Erzeugen der Codetabelle

```
1   // 4. Generate Huffman codes by traversing tree with recursion and DFS
    private static void generateCodes(Node node, String currentCode, Map<Character, String>
        codeTable) {
3       // Base case
        if (node == null) {
5           return;
        }
7
        // If it is a leaf, it represents a character, then assign a code
9       if (node.isLeaf()) {
            if (node.character != null) {
11              codeTable.put(node.character, currentCode);
            }
13          // Leaf is reached, stop recursion
            return;
15      }

17      // Assign codes 0 until k-1 to children
        for (int i = 0; i < node.children.size(); i++) {
19          generateCodes(node.children.get(i), currentCode + i, codeTable);
        }
21  }
```

5. Berechnung der Gesamtlänge

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

6. Ausgabe

```
    // Print code table in a readable format
2   private static void printCodeTable(Map<Character, String> codeTable, Map<Character,
        Long> frequencyMap) {
        if (codeTable.isEmpty()) {
4           System.out.println("{}");
            return;
6       }
        List<Character> sortedKeys = new ArrayList<>(codeTable.keySet());
```

```
8       // Sorting is optional
        // Sort by length of code
10      sortedKeys.sort((a, b) -> Integer.compare(codeTable.get(a).length(),
        codeTable.get(b).length()));
        // Sort by frequency
12      sortedKeys.sort((a, b) -> Long.compare(frequencyMap.get(b), frequencyMap.get(a)));

14      System.out.println("{");
        for (Character character : sortedKeys) {
16          // Handle special characters for printing
            String c;
18          c = switch (character) {
                case '␣' -> "␣";
20              case '"' -> c = "\"";
                default -> String.valueOf(character);
22          };
            System.out.println("␣␣'" + c + "':␣" + codeTable.get(character) + "␣(Freq:␣"
24                  + frequencyMap.get(character) + ")");
        }
26      System.out.println("}");
    }
```

2 Teilaufgabe B

2.1 Lösungsidee

Verständnis der Aufgabe

Die Grundidee dieser Aufgabe bleibt gleich: Ein Unicode-Text soll mit Hilfe von Perlen kodiert werden. Allerdings haben die Perlen nun einen unterschiedlichen Durchmesser. Ziel ist es nun, die Gesamtlänge der resultierenden Perlenkette, also die kodierte Nachricht, zu minimieren.
Wenn dieses Problem weiterhin als Optimierungsproblem zu Erstellung einer optimalen, präfixfreien Kodierung (wie bereits im ersten Teil der Aufgabe) betrachtet wird, führt die Einführung variabler Perlendurchmesser zu einer Verallgemeinerung des ursprünglichen Problems. Denn in dieser Erweiterung entspricht der Durchmesser jeder Perle den Kosten des jeweiligen Symbols im Codealphabet. Im Folgenden wird einfachheitshalber, die Perle als Symbol bezeichnet, da diese verwendet werden, um den Text zu kodieren und der Durchmesser der Perle wird als Kosten des Symbols bezeichnet.

Diese Variante ist bekannt als das Problem der **präfixfreien Kodierung mit ungleichen Symbolkosten** (prefix-free coding with unequal letter cost). Im Gegensatz zur klassischen Huffman-Kodierung, die von gleichen Symbolkosten ausgeht, ist diese Verallgemeinerung deutlich komplexer. Es ist bis heute **nicht bekannt**, ob dieses Problem in **P** liegt oder **NP-schwer** ist.

Überlegungen zum Lösungsansatz

Die aus Kapitel 1 implementierte n-näre Huffman-Kodierung scheitert bei dieser Problemstellung, da die Minimierung der Anzahl der Kodierungsbuchstaben nicht notwendigerweise zur Minimierung der Gesamtkosten führt (die Kosten werden nämlich nicht berücksichtigt).
Im Folgenenden werden drei Strategien zur Lösung dieses Problems untersucht:

2.2 Strategie 1: Heuristische n-näre Huffman Modifikation mit greedy Kostenzuweisung

2.2.1 Kernidee

Dieser Ansatz ist vermutlich der Intuitivste und am wenigsten aufwendige und ist in zwei Schritte aufgeteilt:

1. **Erstellung eines n-nären Huffman Baums**
   Zunächst wird ein n-närer Huffman mit dem aus Kapitel 1 beschriebenen unveränderten Algorithmus erstellt, bei der der Baum nur basierend auf den Symbolhäufigkeiten gebaut wird. Dieser Schritt ignoriert also die Symbolkosten und bestimmt nur die Topologie des Baums.

2. **Greedy Zuweisung der Kodierungsbuchstaben anhand der Kosten**
   Bei der Traversierung des Baums zur Erstellung der Codewörter erfolgt die Beschriftung der Kanten (von inneren Knoten mit n Kindern) standardmäßig mit den Ziffern bzw. Kodierungszeichen 0 bis $n-1$.
   Während bei der klassischen Methode aus Kapitel 1 diese Zuweisung oft der Reihenfolge der Kinder folgt (z.B. von links nach rechts), ändert sich dies in dieser Heuristik:
   Denn nun erfolgt diese Zuweisung greedy nach folgendem Prinzip:
   Die Kante, die zum Teilbaum mit der höchsten Häufigkeit führt, erhält jetzt den Kodierungsbuchstaben mit den geringsten Kosten. Die zweithäufigste Kante erhält das zweigünstigste Symbol usw. Dadurch wird angestrebt, dass teure Kodierungszeichen, also Perlen mit großem Durchmesser möglichst selben verwendet werden (in Pfaden mit seltenen Symbolen). So werden günstige Zeichen möglichst früh im Baum platziert, sodass sie häufiger vorkommen und damit mehr zur Gesamtkodierung beitragen können.

Diese lokale Regel zur Zuordnung führt nicht zu einer global optimalen Lösung, aber sie verteilt die teuren Kosten möglichst kosteneffizient in dem bestehenden Baum. Diese Heuristik ist leicht implementierbar, da diese nur eine kleine Anpassung der n-nären Huffman-Kodierung benötigt. Aus diesen Gründen wurde diese Strategie als erster Ansatz für die Lösung dieses Teilproblems gewählt. Sie dient als Baseline, an der weiterführende Verfahren verglichen werden können, die diese Strategie idealerweise übertreffen.

2.2.2 Umsetzung

Die Umsetzung dieser Strategie erfordert nur eine Anpassung der Teilschritte 4 und 5 des beschriebenen Algorithmus aus Kapitel 1. Die Schritte 1 bis 3 bleiben identisch (bis auf die Einlesung der Daten in Schritt 1 die minimal angepasst wird). Folgendes wird angepasst:

4. **Traversierung mit greedy Kostenzuweisung**
   Die eigentliche Traversierung des Baums mit der rekursiven Tiefensuche bleibt identisch. Für die Zuweisung der Kosten, werden zunächst die Kinder nach ihrer Häufigkeit absteigend sortiert. Die verfügbaren Kodierungssymbole (in diesem Fall die Ziffern 0 bis $n-1$), werden auch gemäß ihren zugehörigen Kosten aufsteigend sortiert. Mit einer Schleife über alle Kinder, wird den Kanten zu den Teilbäumen nun jeweils das günstigste noch verfügbare Symbol zugewiesen, beginnend mit dem häufigsten Teilbaum. Für jedes Kind wird dieser Prozess rekursiv wiederholt, wobei der aktuelle Pfad (die bisher zugewiesenen Kodierungssymbole) jeweils an das entstehende Codewort angehängt wird.

5. **Berechnung der Gesamtlänge der kodierten Nachricht**
   In diesem Schritt wird die Gesamtlänge der kodierten Nachricht bestimmt. Im Unterschied zur klassischen Huffman-Kodierung, bei der die Länge eines Codeworts einfach durch die Anzahl der Symbole bestimmt wird, müssen hier die individuellen Kosten der verwendeten Kodierungssymbole mit einberechnet werden. Die Gesamtlänge ergibt sich also nicht nur aus der Anzahl der Symbole im Codewort, sondern aus der Summe der Kosten aller Symbole, die zur Kodierung eines Zeichens verwendet werden.

2.2.3 Komplexitätsanalyse

Die Erklärung für die unveränderten Teilschritten bleibt wie in den Kapitel 1 und 2.2 gleich. Die Komplexität des Algorithmus has sich wie folgt, geändert:

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	$\mathcal{O}(k \cdot n \log n)$	$\mathcal{O}(k \cdot n)$
Berechnung der Gesamtlänge	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Ausgabe Codetabelle (mit Sortierung)	$\mathcal{O}(k \log k)$	$\mathcal{O}(k)$
Gesamt	$\boldsymbol{\mathcal{O}(L + k \log k)}$	$\boldsymbol{\mathcal{O}(L + k)}$

Table 2: Laufzeit- und Speicherkomplexität Teilaufgabe B Strategie 1

- **Codetabelle:** Dieser Schritt ist durchläuft jeden Knoten (insgesamnt k Knoten) genau einmal mit DFS (Tiefensuche). Pro Knoten werden die Kinder und Kodierungszeichen sortiert, was in $\mathcal{O}(n \log n)$ ist. Daher beträgt die Laufzeit für diesen Schritt $\mathcal{O}(k \cdot n \log n)$.
- **Gesamtlänge:** Dieser Schritt iteriert über alle verschiedenen Symbole, die einen Code bekommen. Es wird über jedes Zeichen in Code iteriert und somit ist dieser Schritt in $\mathcal{O}(k \cdot L_{max})$. L_{max} ist hier die maximale Code-Länge (konstant), daher $\mathcal{O}(k)$.

2.2.4 Beispiele

Hier die Ausgaben des Programmes zu den Beispielen auf der BwInf-Webseite. Mit Ausnahme der Datei *schmuck5.txt* (für die ein offizieller Vergleichswert vorliegt), wurde die Konsolenausgabe auf die Gesamtlänge der kodierten Nachricht reduziert. Dies wurde gemacht für eine klarere Vergleichbarkeit der Ergebnisse zwischen den noch folgenden Algorithmen.
Die gesamte Ausgabe ist hier **AusgabenBStrategie1.txt**

EDIT

zufinden. Die Ausgabe wird zu dem finalen (und besten) Programm ausführlicher gegeben.

1. **schmuck1.txt**

```
  ...
Gesamtlänge der Botschaft 197 (Anzahl in Perlen) bzw. 19.7cm
```

2. **schmuck2.txt**

```
  ...
Gesamtlänge der Botschaft 145 (Anzahl in Perlen) bzw. 14.5cm
```

3. **schmuck3.txt**

```
  ...
Gesamtlänge der Botschaft 279 (Anzahl in Perlen) bzw. 27.9cm
```

4. **schmuck4.txt**

```
  ...
Gesamtlänge der Botschaft 154 (Anzahl in Perlen) bzw. 15.4cm
```

5. **schmuck5.txt**

```
Codetabelle:
(Jede Ziffer steht für eine Perlenfarbe z.B. könnte '0' rot bedeuten und '1' blau)
{
  '␣': 2 (Freq: 151, Länge: 2)
  'e': 3 (Freq: 110, Länge: 3)
```

```
  't': 4 (Freq: 71, Länge: 4)
  'i': 5 (Freq: 67, Länge: 5)
  'o': 00 (Freq: 64, Länge: 2)
  's': 01 (Freq: 61, Länge: 2)
  'a': 02 (Freq: 58, Länge: 3)
  'n': 03 (Freq: 56, Länge: 4)
  'r': 04 (Freq: 51, Länge: 5)
  'c': 05 (Freq: 44, Länge: 6)
  'd': 06 (Freq: 37, Länge: 7)
  'l': 10 (Freq: 33, Länge: 2)
  'm': 11 (Freq: 27, Länge: 2)
  'h': 12 (Freq: 26, Länge: 3)
  'u': 14 (Freq: 25, Länge: 5)
  'f': 16 (Freq: 20, Länge: 7)
  'p': 15 (Freq: 20, Länge: 6)
  'b': 60 (Freq: 16, Länge: 7)
  'y': 61 (Freq: 13, Länge: 7)
  'g': 62 (Freq: 11, Länge: 8)
  '.': 63 (Freq: 8, Länge: 9)
  'q': 66 (Freq: 5, Länge: 12)
  ',': 130 (Freq: 5, Länge: 5)
  'v': 131 (Freq: 5, Länge: 5)
  'x': 132 (Freq: 5, Länge: 6)
  'w': 133 (Freq: 4, Länge: 7)
  'F': 134 (Freq: 3, Länge: 8)
  'T': 135 (Freq: 3, Länge: 9)
  'A': 650 (Freq: 1, Länge: 12)
  'G': 644 (Freq: 1, Länge: 14)
  'H': 651 (Freq: 1, Länge: 12)
  'S': 640 (Freq: 1, Länge: 11)
   ''': 645 (Freq: 1, Länge: 15)
  'j': 652 (Freq: 1, Länge: 13)
  '-': 643 (Freq: 1, Länge: 13)
  '1': 646 (Freq: 1, Länge: 16)
  '2': 136 (Freq: 1, Länge: 10)
  '5': 653 (Freq: 1, Länge: 14)
  '9': 654 (Freq: 1, Länge: 15)
  'z': 641 (Freq: 1, Länge: 11)
  ';': 642 (Freq: 1, Länge: 12)
}

Gesamtlänge der Botschaft 4010 (Anzahl in Perlen) bzw. 401.0cm
```

Dieses Ergebnis (4010) ist ca. 26.82% schlechter als der Vergleichswert (3162) von der BwInf Webseite, daher ist hier noch viel Platz für Verbesserung.

6. **schmuck6.txt**

```
  ...
Gesamtlänge der Botschaft 244 (Anzahl in Perlen) bzw. 24.4cm
```

7. **schmuck7.txt**

```
  ...
Gesamtlänge der Botschaft 153144 (Anzahl in Perlen) bzw. 15314.4cm
```

8. **schmuck8.txt**

```
  ...
Gesamtlänge der Botschaft 3615 (Anzahl in Perlen) bzw. 361.5cm
```

9. **schmuck9.txt**

```
...
Gesamtlänge der Botschaft 41056 (Anzahl in Perlen) bzw. 4105.6cm
```

Dieses Ergebnis (41056) ist ca. 12.18% schlechter als der Vergleichswert (36597) von der BwInf Webseite, daher ist hier noch etwas Platz für Verbesserung.

2.2.5 Quellcode

Wichtige Teile des Quellcodes:

4. Erzeugen der Codetabelle mit greedy Heuristik

```java
// 4. Generate Huffman codes with cost-aware digit assignment
private static void generateCodesCostAware(Node node, String currentCode, Map<Character,
    String> codeTable,
        List<Integer> costs) {
    // 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.isEmpty() ? "0" : currentCode);
        }
        // Leaf is reached, stop recursion
        return;
    }

    List<Node> children = node.children;

    // 1. Sort children by frequency - Descending (made copy to not modify the main
    // list, not sure if needed)
    List<Node> sortedChildren = new ArrayList<>(children);
    sortedChildren.sort(Comparator.comparingLong(Node::getFrequency).reversed());

    // 2. Prepare digits sorted by cost - Ascending
    // DigitCost is a wrapper to map each digit with its corresponding cost
    List<DigitCost> digitCosts = new ArrayList<>();

    for (int i = 0; i < costs.size(); i++) {
        digitCosts.add(new DigitCost(i, costs.get(i)));
    }
    // Sort by cost ascending, for this input not needed as digits are already sorted
    Collections.sort(digitCosts);

    // 3. Assign cheapest digits to most frequent children
    for (int i = 0; i < sortedChildren.size(); i++) {
        Node child = sortedChildren.get(i);
        int assignedDigit = digitCosts.get(i).digit; // i-th cheapest digit
        generateCodesCostAware(child, currentCode + assignedDigit, codeTable, costs);
    }
}
```

5. Berechnung der Gesamtlänge

```java
// 5. Calculate total cost of encoded message
private static long calculateTotalCost(Map<Character, String> codeTable, Map<Character,
    Long> frequencyMap,
        List<Integer> costs) {
    long totalCost = 0;

    // Iterate over each character
    for (Map.Entry<Character, Long> entry : frequencyMap.entrySet()) {
        char character = entry.getKey();
        long frequency = entry.getValue();
```

```
            String code = codeTable.get(character);

            long costOfCode = 0;
            for (char digitChar : code.toCharArray()) {
                int digit = Character.getNumericValue(digitChar);
                costOfCode += costs.get(digit);
            }
            totalCost += frequency * costOfCode;
        }
        return totalCost;
    }
```

2.3 Strategie 2: Modifizierung Golin/Li Algorithmus

2.3.1 Kernidee

Da die erste implementiere Strategie als Referenz verwendet wird und die berechneten Gesamtlängen der kodierten Nachrichten nicht nahe genug an den Werten der BwInf Webseite liegen, wird eine weiterer Algorithmus erkundet. Das Problem einen Text nahezu optimal präfixfrei zu kodieren, wenn die Symbole zur Kodierung unterschiedliche Kosten aufweisen, ist nicht trivial. Der Algorithmus von Golin und Li [2] bietet eine effiziente $\mathcal{O}(n \log n)$ Lösung (mit n als Anzahl der Quellsymbole) mit einer bewiesenen guten Fehlerschranke gegenüber dem Optimum, die besser als der Ansatz aus Kapitel 2.2 sein sollte. Daher wird im Folgenden ein Algorithmus basierend auf [2, Fig 6. und Kapitel 4] implementiert, der die Kernprinzipien des Golin/Li-Algorithmus anwendet, um einen solchen Präfixcode zu finden. Der umgesetzte Algorithmus vereinfacht allerdings Teile aus dem Paper etwas, da diese sehr komplex ist. Daher kann die bewiesene Fehlerschranke zu dem Optimum nicht mehr garantiert werden. Dennoch sollte der Algorithmus bessere Resultate bringen.
Zum Verständnis des Algorithmus werden folgende zwei Hauptdatensätze definiert:

1. Eine Menge von n Quellsymbolen, jeweils mit einer zugehörigen Auftrittswahrscheinlichkeit $0 < p < 1$, basierend auf deren Häufigkeit im Text. Diese Symbole werden absteigend nach ihrer Wahrscheinlichkeit sortiert ($p_1 \geq p_2 \geq ... \geq p_n$).

2. Eine Menge von t Kodierungsbuchstaben (Zielalphabet, also Perlenfarben bzw. Ziffern), jeweils mit positiven Kosten c_m (also Durchmesser der Perle). Diese werden aufsteigend nach ihren Kosten sortiert ($c_1 \leq c_2 \leq ... \leq c_t$)

Ein wichtiger Aspekt zur Lösung des Problems ist, die unterschiedlichen Kosten der verfügbaren Kodierungsbuchstaben fair den Quellsymbolen zuzuordnen, welche unterschiedlich oft auftreten. Anstatt wie der klassische Huffman-Algorithmus das Zusammenfügen der unwahrscheinlichsten Symbole und dann eine Zuordnung mit Kodierungsbuchstaben, wählt Golin/Li einen globaleren Ansatz, der auf eine intelligente Aufteilung des Wahrscheinlichkeitsraums basiert.
Die Kernidee des Algorithmus ist eine rekursive Aufteilung der Quellsymbole in sogenannte "Bins" bzw. Gruppen. Die Zuweisung des ersten Kodierungsbuchstabens zu verschiedenen Gruppen von Quellsymbolen soll dabei nicht willkürlich, sondern kostenbewusst gemacht werden. Man möchte erreichen, dass Kodierungsbuchstaben mit *geringen* Kosten tendenziell den Gruppen von Quellsymbolen zugeordnet werden, die zusammen eine höhere Gesamtwahrscheinlichkeit aufweisen. Dies ist intuitiv.
Um dieses Ausbalancieren zu erreichen, betrachtet der Algorithmus nicht die einzelnen Symbole isoliert, sondern ihre Verteilung im **kumulativen Wahrscheinlichkeitsraum**. an kann sich vorstellen, dass alle Quellsymbole, sortiert nach ihrer Wahrscheinlichkeit p_m auf einem Intervall von 0 bis 1 angeordnet sind, wo die Länge des Abschnitts für Symbol i seiner Wahrscheinlichkeit p_m entspricht. Die Gesamtlänge dieses Intervalls ist 1.

Der Algorithmus zerlegt nun dieses Gesamtintervall (bzw. in rekursiven Schritten ein Teilintervall, einer bereits gebildeten Gruppe) in t Segmente, wo t die Anzahl der verfügbaren Kodierungsbuchstaben ist. Jedes Segment wird einem Kodierungsbuchstaben m zugeordnet und stellt ein Bin dar.
Nun ist das Entscheidende, wie groß diese Bins/Segmente gemacht werden. Hierfür wird die **charakteristische Wurzel c** benutzt. Sie ist gewissermaßen ein Skalierungsfaktor, der die Kosten c_m von jedem Kodierungsbuchstaben in ein "effektives Gewicht" (definiert als $2^{-c \cdot c_m}$) übersetzt. Dieses Gewicht repräsentiert den "fairen Anteil", die dieser Buchstabe in der Kodierung beiträgt. Die Breite des Bins für den Kodierungs-

buchstaben m wird nun proportional zu diesem effektiven Gewicht festgelegt:

$$\text{Breite(Bin } m) = w \text{ (Gesamtwahrscheinlichkeit der aktuellen Symbolgruppe)} \cdot 2^{-c \cdot c_m} \tag{3}$$

Diese charakteristische Wurzel c ist definiert als eindeutige positive reelle Zahl, die die Gleichung mit t: Zahl der Kodierungssymbolen und c_m die Kosten des m-ten Symbols.

$$\sum_{m=1}^{t} 2^{-c \cdot c_m} = 1 \tag{4}$$

erfüllt. Diese Gleichung ist eine Verallgemeinerung der Kraft-McMillan-Ungleichung für den Fall ungleicher Kosten und ist eine notwendige Bedingung für die Existenz eines vollständigen Präfixcodes.

Aus der Definition von c folgt, dass die Summe dieser Gewichte 1 betragen und diese Tatsache ist auch mathematisch notwendig, um den gesamten Wahrscheinlichkeitsraum der Quellsymbole abzudecken. In dem Fall mit den Perlen heißt das konkret, dass eine Perle mit großem Durchmesser durch das c ein kleines "Gewicht", also einen **kleineren Anteil** und eine kleine Perle ein großes Gewicht, also einen **größeren Anteil** am Wahrscheinlichkeitsintervall bekommt.

Nachdem das Wahrscheinlichkeitsintervall mit den gewichteten Kosten im Blick, aufgeteilt wurde, werden die einzelnen Quellsymbole diesen Bins zugeordnet. Dies geschieht mit einer Heuristik: Für jedes Symbol p_i wird sein **Wahrscheinlichkeitsmittelpunkt**

$$s_i = P_{i-1} + \frac{p_i}{2} \tag{5}$$

betrachtet. Das Symbol wird dem Bin m zugewiesen, in dessen Intervall $[L_m, R_m)$ dieser Mittelpunkt fällt. Dieser Prozess des Zuweisens von Symbolen an Bins wird **rekursiv** für jede dieser Bins wiederholt, um den zweiten Kodierungsbuchstaben zu bestimmen usw. In jedem rekursiven Schritt wird das relevante Teilintervall der Wahrscheinlichkeit erneut nach dem gleichen Prinzip (also kosten-gewichtet mit c) aufgeteilt. Dadurch entsteht ein Codebaum von dem die Struktur die ungleichen Kosten **von Anfang an berücksichtigt**, im Gegensatz zu n-närem Huffman. Daher sollte dieser Algorithmus bessere Ergebnisse liefern, die aber noch garantiert nicht optimal sind. Dieser Baum muss am Ende nicht mehr traversiert werden, da der Baum "top-down" (im Gegensatz zu Huffman) erstellt wird, und wenn der rekursive Aufruf zur Erstellung des Baums an ein "Blatt" (hier also wenn in einem Bin nur ein Symbol ist), wird der sich der Pfad bis dahin gemerkt und als Codewort des Symbols festgelegt.

Dieser Aufteilungsmechanismus funktioniert und ist der Kern dieser Strategie (2.3), weil er systematisch die **Wahrscheinlichkeiten der Quellsymbole mit den Kosten der Kodierungsbuchstaben verknüpft**. Die wahrscheinlicheren Symbole (die größere Abschnitte im kumulativen Intervall belegen) haben eine höhere Chance, in den breiteren Bins zu landen. Da die breiteren Bins durch die c-gewichtete Aufteilung den kostengünstigeren Kodierungsbuchstaben entsprechen, bekommen die häufigen Symbolgruppen tendenziell die "billigeren" Kodierungsbuchstaben.

Anmerkung: Die Mittelpunkt Heuristik kann auch zu suboptimalen Verteilungen führen, sodass es z.B. leere Bins zwischen gefüllten Bins sind oder, dass alle Symbole in Bin eins sind. Das Golin/Li Paper definiert sogenannte "Left Shift" und "Right Shift" Operationen, um diese Verteilungen zu verbessern. Der Left Shift füllt leere Bins auf, indem Symbola aus den nachfolgende Bins nach vorne verschoben werden. Dieser Prozess ist in dem Paper allerdings sehr komplex und wird in dieser Implementierung stark vereinfacht. Der Right Shift behandelt den seltenen Spezialfall, dass alle Symbole im ersten Bin landen, indem das letzte Symbol in den zweiten Bin zu verschoben wird. So wird eine übermäßige Nutzung des ersten Bins vermieden.

2.3.2 Umsetzung

Diese Implementierung folgt den Kernprinzipien des Algorithmus von Golin/Li [2, Fig. 6 und Kapitel 4], Kapitel IV, wobei die Logik des Left Shifts vereinfacht wurde:

Der Schritt 1 - Einlesung der Daten und der Schritt 2 – Häufigkeitsanalyse der Symbole im Text, aus Kapitel 1 und 2.2 bleibt gleich. Für kommende Berechnungen mit den Durchmessern der Perlen werden diese allerdings als Doubles gespeichert.

3. **Berechnung der Charakteristischen Wurzel** c
   Die Bestimmung der charakteristischen Wurzel c ist ein wichtiger Teilschritt, der für die Aufteilung in Bins benötigt wird. Um c numerisch zu berechnen, wird die Gleichung 4 gelöst, indem das Bisektionsverfahren verwendet wird. Das funktioniert, indem es schrittweise das Intervall $[0, 10]$ halbiert und jeweils entscheidet, ob die Nullstelle der Funktion $f(c) = \sum 2^{-c \cdot c_m} - 1$ halbiert und jeweils entscheidet, ob die Nullstelle der Funktion $c < 10$ da das das gewählte Startintervall ist.

4. **Rekursive Codegenerierung**
   Für die Rekursion muss jeweils die linke Grenze (bezeichnet als l) und die rechte Grenze (bezeichnet als r) des betrachteten Intervalls bekannt sein.

   0. **Basisfall**
      Die Rekursion endet, wenn das betrachtete Teilintervall nur noch ein einziges Symbol enthält ($l == r$). Der bis dahin akkumulierte Präfix U (Pfad bis zum Bin) wird diesem Symbol als finales Codewort in der Codetabelle zugeordnet

   1. **Symbolzuweisung zu Initial-Bins**
      Jedes Symbol i innerhalb der aktuellen Teilmenge (l bis r) wird basierend auf seinem Wahrscheinlichkeitsmittelpunkt berechnet mit 5 zu dem Bin m zugeordnet. Die Indizes der Symbole werden in *initialBins* gesammelt.

   2. **Left Shift (Stark vereinfacht)**
      Anstatt der komplexen Shifting Operation aus dem Originalalgorithmus implementiert der Code einen vereinfachten Left Shift. Er iteriert durch die *initialBins*. Wenn ein Bin leer ist, wird versucht, ihn mit dem nächsten noch nicht zugewiesenen Symbol aus der sortierten Liste zu füllen. So wird die Idee, Lücken zu schließen, versucht zu bewahren.

   3. **Right Shift**
      Der spezifische Fall, dass alle Symbole im ersten Bin landen (finalBins.size() == 1), wird explizit überprüft. Wenn dies zutrifft und mehr als ein Symbol vorhanden ist, wird der Index des letzten Symbols aus dem ersten Bin entfernt und einem neu erstellten zweiten Bin hinzugefügt.

   4. **Rekursiver Aufruf**
      Schließlich wird für jeden nicht leeren Bin ein rekursiver Aufruf gestartet. Der Codierungsbuchstabe des Bins wird dabei an den aktuellen Präfix U angehängt, damit der Basisfall das Codewort festlegen kann. Der rekursive Aufruf erfolgt dann für die Teilmenge der Symbole in diesem Bin mit den angepassten Grenzen für das Betrachten des Intervalls.

Der Schritt 5 – Berechnung der Gesamtlänge des kodierten Texts und Schritt 6 – Ausgabe der Codetabelle, aus Kapitel 1 und 2.2 bleibt gleich.

2.3.3 Komplexitätsanalyse

Nun wird die Komplexität der einzelnen Schritte des Algorithmus betrachtet, die Erklärungen für die unveränderten Teilschritten bleiben wie in den Kapitel 1 und 2.2 gleich. Dabei ist:

- L die Länge des Eingabetextes,
- k die Anzahl der verschiedenen Zeichen (eindeutige Quellsymbole),
- n die Anzahl der Kodierungsbuchstaben (Anzahl Perlenfarben).

Schritt	**Laufzeit**	**Speicherbedarf**
Einlesen der Datei	$\mathcal{O}(L)$	$\mathcal{O}(L)$
Häufigkeitsanalyse	$\mathcal{O}(L)$	$\mathcal{O}(k)$
Berechnung der Charakteristischen Wurzel	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Rekursive Codegenerierung	$\mathcal{O}(k \log k)$	$\mathcal{O}(k \log k + k + n)$
Berechnung der Gesamtlänge	$\mathcal{O}(k)$	$\mathcal{O}(1)$
Ausgabe Codetabelle (mit Sortierung)	$\mathcal{O}(k \log k)$	$\mathcal{O}(k)$
Gesamt	$\boldsymbol{\mathcal{O}(L + k \log k)}$	$\boldsymbol{\mathcal{O}(L + k + n + k \log k)}$

Table 3: Laufzeit- und Speicherkomplexität Teilaufgabe B mit Strategie 2

- **Berechnung der Charakteristischen Wurzel c:** Bei jedem Aufruf der Funktion $f(c)$ muss die Summe $\sum_{i=1}^{n} 2^{-c \cdot c_m}$ berechnet werden. Dies braucht t Potenzierungs- und Additionsoperationen. Die Laufzeit für eine einzelne Auswertung von $f(c)$ ist daher $\mathcal{O}(n)$. Da dies 100-mal getan wird (Bisektionsverfahren 100-mal angewendet), gilt $\mathcal{O}(100 \cdot n) = \mathcal{O}(n)$
- **Rekursive Codegenerierung:** Die rekursive Methode *generateCodesGolinLi()* wird $\mathcal{O}(k)$ mal aufgerufen. Die zeitaufwendigste Operation in jedem Aufruf ist die Zuweisung von k' Symbolen zu n Bins, was $\mathcal{O}(n \cdot k')$ braucht. Die Gesamtlaufzeit hängt von der Rekursionstiefe und die Balance des Codebaums ab. Für diese Implementierung mit vereinfachtem Shifting ergibt sich daher geschätzt eine Laufzeit zwischen $\mathcal{O}(n \cdot k \log k)$ (balancierter Baum, aufgrund Aufteilung), da n eine kleine Konstante ist: $\mathcal{O}(k \log k)$.

Insgesamt ist die Laufzeit dominiert durch die rekurive Struktur des Algorithmus und die optionale Sortierung für die Ausgabe der Codetabelle. Der Speicherbedarf (hier geschätzt) ist etwas größer als bei Strategie 1, dennoch nicht bemerkenswertes. Die tatsächliche Speicherkomplexität hängt stark von der Struktur des generierten Codes ab und der Tiefe des Rekursionsstacks.
Wenn die Liste mit n Kodierungsbuchstaben nicht sortiert wäre, wie in den Beispielen der BwInf Webseite, müsste diese noch sortiert werden und die Gesamtlaufzeit erhöht sich um $\mathcal{O}(n \log n)$. Allerdings ist n oft sehr klein, was also keinen großen Unterschied macht. Daher erweist sich dieser Algorithmus auch bei größeren Texten als leistungsfähig und mit derselben asymptoischen Laufzeit wie Strategie 1.

2.3.4 Beispiele

Hier die Ausgaben des Programmes zu den Beispielen auf der BwInf-Webseite. Mit Ausnahme der Datei *schmuck5.txt* (für die ein offizieller Vergleichswert vorliegt), wurde die Konsolenausgabe auf die Gesamtlänge der kodierten Nachricht reduziert. Dies wurde gemacht für eine klarere Vergleichbarkeit der Ergebnisse zwischen den noch folgenden Algorithmen.
Die gesamte Ausgabe ist hier **AusgabenBStrategie2.txt**

EDIT

zufinden. Die Ausgabe wird zu dem finalen (und besten) Programm ausführlicher gegeben.

1. **schmuck1.txt**

```
...
Gesamtlänge der Botschaft 195 (Anzahl in Perlen) bzw. 19.5cm
```

2. **schmuck2.txt**

```
...
Gesamtlänge der Botschaft 135 (Anzahl in Perlen) bzw. 13.5cm
```

3. **schmuck3.txt**

```
...
Gesamtlänge der Botschaft 287 (Anzahl in Perlen) bzw. 28.7cm
```

4. **schmuck4.txt**

```
  ...
Gesamtlänge der Botschaft 137 (Anzahl in Perlen) bzw. 13.7cm
```

5. **schmuck5.txt**

```
Charakteristische Wurzel c = 1.386461

Codetabelle (Golin/Li):
{
  '␣': 00 (Freq: 151, Länge: 2)
  'e': 010 (Freq: 110, Länge: 3)
  't': 011 (Freq: 71, Länge: 3)
  'i': 02 (Freq: 67, Länge: 3)
  'o': 100 (Freq: 64, Länge: 3)
  's': 101 (Freq: 61, Länge: 3)
  'a': 110 (Freq: 58, Länge: 3)
  'n': 111 (Freq: 56, Länge: 3)
  'r': 112 (Freq: 51, Länge: 4)
  'c': 12 (Freq: 44, Länge: 3)
  'd': 13 (Freq: 37, Länge: 4)
  'l': 200 (Freq: 33, Länge: 4)
  'm': 201 (Freq: 27, Länge: 4)
  'h': 210 (Freq: 26, Länge: 4)
  'u': 211 (Freq: 25, Länge: 4)
  'f': 22 (Freq: 20, Länge: 4)
  'p': 23 (Freq: 20, Länge: 5)
  'b': 30 (Freq: 16, Länge: 4)
  'y': 310 (Freq: 13, Länge: 5)
  'g': 311 (Freq: 11, Länge: 5)
  '.': 312 (Freq: 8, Länge: 6)
  ',': 32 (Freq: 5, Länge: 5)
  'q': 33 (Freq: 5, Länge: 6)
  'v': 400 (Freq: 5, Länge: 6)
  'x': 401 (Freq: 5, Länge: 6)
  'w': 410 (Freq: 4, Länge: 6)
  'T': 42 (Freq: 3, Länge: 6)
  'F': 411 (Freq: 3, Länge: 6)
  'A': 43 (Freq: 1, Länge: 7)
  'G': 44 (Freq: 1, Länge: 8)
  '2': 52 (Freq: 1, Länge: 7)
  '5': 53 (Freq: 1, Länge: 8)
  '9': 60 (Freq: 1, Länge: 7)
  'z': 61 (Freq: 1, Länge: 7)
  ';': 62 (Freq: 1, Länge: 8)
  'H': 500 (Freq: 1, Länge: 7)
  'S': 501 (Freq: 1, Länge: 7)
   ''': 502 (Freq: 1, Länge: 8)
  'j': 510 (Freq: 1, Länge: 7)
  '-': 511 (Freq: 1, Länge: 7)
  '1': 512 (Freq: 1, Länge: 8)
}

Gesamtlänge der Botschaft 3374 (Anzahl in Perlen) bzw. 337.4cm
```

Dieses Ergebnis (3374) ist ca. 6.7% schlechter als der Vergleichswert (3162) von der BwInf Webseite, also bereits eine großer Fortschritt im Verleich zu Strategie 1.

6. **schmuck6.txt**

```
...
Gesamtlänge der Botschaft 240 (Anzahl in Perlen) bzw. 24.0cm
```

7. **schmuck7.txt**

```
...
Gesamtlänge der Botschaft 144964 (Anzahl in Perlen) bzw. 14496.4cm
```

8. **schmuck8.txt**

```
...
Gesamtlänge der Botschaft 3433 (Anzahl in Perlen) bzw. 343.3cm
```

9. **schmuck9.txt**

```
...
Gesamtlänge der Botschaft 38516 (Anzahl in Perlen) bzw. 3851.6cm
```

Dieses Ergebnis (38516) ist ca. 5.2% schlechter als der Vergleichswert (36597) von der BwInf Webseite, daher eigentlich schon akzeptabel.

2.3.5 Quellcode

Wichtige Teile des Quellcodes:

Berechnung der Charakteristische Wurzel c

```java
// 3. Find Characteristic root c for equation: sum(2^(-c*d_i)) = 1
private static double findCharacteristicRoot(double[] diameters) {
    // Define function f(c) = sum(2^(-c*d_i)) - 1
    Function<Double, Double> f = (c) -> {
        double sum = 0.0;
        for (double d : diameters) {
            sum += Math.pow(2.0, -c * d);
        }
        return sum - 1.0;
    };

    // Find c, numberic approach with bisection
    double low = 0.0;
    double high = 10.0; // Assumption: c not too big
    // Check limits
    if (f.apply(TOLERANCE) < 0)
        return Double.NaN; // No positiv root possible

    // I randomly chose 100 iterations, probably good enough
    for (int i = 0; i < 100; i++) {
        double mid = low + (high - low) / 2.0;
        double f_mid = f.apply(mid);

        if (Math.abs(f_mid) < TOLERANCE) {
            return mid;
        } else if (f_mid > 0) { // Root on the right
            low = mid;
        } else { // Root on the left
            high = mid;
        }
    }
    // Return best value
    return low + (high - low) / 2.0;
}
```

Rekursive Codegenerierung

```java
// 4. Recursive method for code generation with pseudocode from Golin/Li as basis
// (Fig.6 in paper)
// U is the current prefix code
private static void generateCodesGolinLi(List<Symbol> symbols, int l, int r,
        String U, Map<Character, String> codeTable,
        List<Perle> sortedDiameters, double cRoot) {

    // 0. Base case, only one symbol in [l, r]
    if (l == r) {
        Symbol sym = symbols.get(l);
        codeTable.put(sym.character, U);
        return;
    }
    if (l > r)
        return;

    // 1. Create initial bins Im*
    int k = sortedDiameters.size();
    // List of List with original indexes
    List<List<Integer>> initialBins = new ArrayList<>(k);
    for (int i = 0; i < k; i++) {
        initialBins.add(new ArrayList<>());
    }

    double P_start = symbols.get(l).cumulativeProbStart; // P_{l-1}
    double P_end = symbols.get(r).cumulativeProbEnd; // P_r
    // Total weight of this recursion step (w(v))
    double w = P_end - P_start;

    // No more weight to distribute
    if (w <= TOLERANCE)
        return;

    // L in paper
    double currentL = P_start;

    // Calculate edges Lm*, Rm* (relative limits inside of w) and assigns symbols
    // Index m is Bin m+1 in Paper (0 to k-1 here)
    for (int m = 0; m < k; m++) {
        Perle diamInfo = sortedDiameters.get(m);
        double binWidth = w * Math.pow(2.0, -cRoot * diamInfo.cost);
        double currentR = currentL + binWidth;

        // Find symbols with middle point in [currentL, currentR)
        for (int i = l; i <= r; i++) {
            Symbol sym = symbols.get(i);
            // Centerpoint s_i = P_{i-1} + p_i / 2
            // Wahrscheinlichkeitsmittelpunkt
            double midPoint = sym.cumulativeProbStart + sym.probability / 2.0;

            // Check if fits to Bin m
            if (midPoint >= currentL - TOLERANCE && midPoint < currentR - TOLERANCE) {
                // Add original index to Bin m
                initialBins.get(m).add(i);
            }
        }
        // Update for next Bin
        currentL = currentR;
    }

    // 2. Very simplified version of left shift from paper
    // Just creates the final bin lists and skips empty bins
    List<List<Integer>> finalBins = new ArrayList<>();
    // Follow index of next available symbol
    int firstItemIndex = l;
    for (int m = 0; m < k; m++) {
        List<Integer> currentBinContent = initialBins.get(m);
        if (currentBinContent.isEmpty()) {
            // Bin is empty: Geht next availalbe symbol
            if (firstItemIndex <= r) {
                // (In paper: find bin where the symbol was originally -> very complex,
I did
                // not manage to implement that correctly :(, so here just add next
```

```
    element to
                // bin to fill it)
74              finalBins.add(Collections.singletonList(firstItemIndex));
                firstItemIndex++;
76          } else {
                // No more elements to fill bin -> bin stays empty (ignored, very sad I
    know)
78          }
        } else {
80          List<Integer> effectiveBinContent = new ArrayList<>();
            for (int originalIndex : currentBinContent) {
82              // Just add if not already used by shifting
                if (originalIndex >= firstItemIndex) {
84                  effectiveBinContent.add(originalIndex);
                }
86          }
            // If bin empty because of skipping, but there are still elements, then just
    use
88          // next
            if (effectiveBinContent.isEmpty() && firstItemIndex <= r) {
90              finalBins.add(Collections.singletonList(firstItemIndex));
                firstItemIndex++;
92          } else if (!effectiveBinContent.isEmpty()) {
                finalBins.add(effectiveBinContent);
94              // Set index for next element after last of this bin
                firstItemIndex = effectiveBinContent.get(effectiveBinContent.size() - 1)
    + 1;
96          }
            // Otherwise bin stays empty
98      }
    }
100 // Check if all elements were assigned
    if (firstItemIndex <= r) {
102     // Elements are still there, that are not inside a bin -> problem somewhere
        // Add to last non empty bin
104     System.out.println(
                "Irgendwas␣broken,␣leck␣mich,␣Elemente␣[" + firstItemIndex + "␣...␣" + r
    + "]␣nicht␣zugeordnet");
106     if (!finalBins.isEmpty()) {
            List<Integer> lastBin = finalBins.get(finalBins.size() - 1);
108         for (int i = firstItemIndex; i <= r; i++) {
                lastBin.add(i);
110         }
        } else {
112         System.out.println("FATAL");
            // Fill first bin, emergency lol
114         List<Integer> onlyBin = new ArrayList<>();
            for (int i = l; i <= r; i++)
116             onlyBin.add(i);
            finalBins.add(onlyBin);
118     }
    }
120
    // 3. Right shift
122 // in paper: if all in bin 1 then move last element to bin 2
    // Here checking this
124 if (finalBins.size() == 1 && finalBins.get(0).size() == (r - l + 1) && k > 1) {
        List<Integer> firstBin = finalBins.get(0);
126     // Only makes sense if more than 1 element in bin
        if (firstBin.size() > 1) {
128         // Remove last index
            int lastElementIndex = firstBin.remove(firstBin.size() - 1);
130         List<Integer> secondBin = new ArrayList<>();
            secondBin.add(lastElementIndex);
132         // Only add if space
            if (finalBins.size() < k) {
134             finalBins.add(secondBin);
            } else {
136             System.out.println("No␣space␣in␣second␣bin");
                firstBin.add(lastElementIndex); // Revert
138         }
        }
140 }
```

```
142     // 4. Recursion for every non empty final bin
        for (int m = 0; m < finalBins.size(); m++) {
144         List<Integer> currentBinIndices = finalBins.get(m);
            if (!currentBinIndices.isEmpty()) {
146             // First index in bin
                int bin_l = currentBinIndices.get(0);
148             // Last index
                int bin_r = currentBinIndices.get(currentBinIndices.size() - 1);
150             // Find color / index for this bin
                // in paper quite complex so here the m-th non empty bin gets the m-th
    cheapest
152             // color
                int colorIndex = sortedDiameters.get(m).digit;
154
                // Recursive call
156             generateCodesGolinLi(symbols, bin_l, bin_r, U + colorIndex, codeTable,
    sortedDiameters, cRoot);
            }
158     }
}
```

2.4 Strategie 3

2.4.1 Umsetzung

2.4.2 Komplexitätsanalyse

2.4.3 Beispiele

2.4.4 Quellcode

2.5 Vergleich der Strategien

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

[2] Mordecai Golin and Li Jian (2007) *More Efficient Algorithms and Analyses for Unequal Letter Cost Prefix-Free Coding*, arXiv:0705.0253 [cs.IT]. https://doi.org/10.48550/arXiv.0705.0253

[3] M. J. Golin and G. Rote (1998) *A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs*, *IEEE Transactions on Information Theory*, 44(5), 1770–1781. https://doi.org/10.1109/18.705558