Abstraktion

Gesucht ist ein Algorithmus, der aus einer gegebenen Häufigkeit verschiedener Buchstaben eine präfixfreie Zuweisung von verschieden gewichteten (anders ausgedrückt, verschieden *teuren*) “Perlen” und deren Kombinationen erstellt, die in diesem Fall zur Codierung des die Buchstaben enthaltenden Textes genutzt werden. Zur Abstraktion des Problems bewegen wir uns von dem Ursprungsgedanken der “Perlen” mit verschiedenen Durchmessern weg und nutzen stattdessen Buchstaben zur Identifizierung der Perlen. Der Durchmesser der Perlen beschreibt hierbei schlichtweg deren “Kosten”, welche es in Hinblick auf den gesamten, zu codierenden Text zu minimieren gilt. Die Menge an verfügbaren Buchstaben mitsamt deren Kosten beschreiben wir fortlaufend als “Encoding-Alphabet”.

Lösungsweg zum Algorithmus

Vorarbeit

Um die bestmöglichen Kombinationen an Perlen für jeden zu codierenden Buchstaben (diese Liste an Zuweisungen wird nun folgend “Encoding” genannt) zu finden, bietet sich die Nutzung eines in der Informatik bereits bekannten Huffman-Baumes an. Dieser wird aus einem Text, genauer gesagt aus der heuristisch ermittelten Häufigkeit jedes einzelnen Buchstaben innerhalb dieses Textes, generiert. Der originale Algorithmus Huffmans funktioniert in mehreren Hinblicken jedoch anders als der hier genutzte Lösungsweg.

Zum einen unterstützt dieser nur das Konstruieren eines binären Huffman-Baumes, was bei unserer Analogie mit den Perlen eine Begrenzung auf exakt 2 verschiedene Perlen – nicht mehr und nicht weniger – zur Folge hätte. Dies gilt es zu verbessern. Zusätzlich vonnöten ist die Berücksichtigung der verschiedenen Perlenkosten. Kurzum, wir suchen einen Algorithmus zum Konstruieren eines n-ären Huffman-Baums mit ungleichen Encodingbuchstaben-Kosten.

Um auf dem ursprünglichen Algorithmus Huffmans aufzubauen, schauen wir zunächst auf dessen Funktionsweise:

1. Analysiere die Häufigkeit jedes Buchstaben in einem gegebenen Text;
2. Jeder Buchstabe sei ein Knoten mit seiner Häufigkeit n;
3. Wiederhole, bis nur noch ein Knoten übrig ist:
    1. Wähle die zwei Knoten der geringsten Häufigkeit und kombiniere diese miteinander zu einem Knoten, der die beiden ursprünglichen Knoten als Kinder besitzt; summiere deren Häufigkeit und weise diese dem übergeordnetem Knoten zu.
    2. Platziere diesen Knoten zurück in die Liste aller verfügbarer Knoten, die es zu kombinieren gilt.
4. Weise jeder Kante des binären Baumes einen von zwei verschiedenen Werten zu, sodass jeder nicht-terminale Knoten jeweils eine Kante mit bspw. einer Null und eine Kante mit einer Eins hat, welche jeweils zu einem anderen Unterknoten führen.

Somit wäre der Huffman-Baum vollständig konstruiert. Damit nun daraus ein Encoding erstellt werden kann, bietet sich an, den Baum rekursiv zu traversieren, sodass jeder Buchstabe der terminalen Knoten (engl. “*Leaf Nodes*”, also Knoten, die keine Kinder haben) einen klar definierten Weg über die Kanten des Binärbaumes hat, wodurch dieser erreicht werden kann. Da wir zuvor diesen Kanten unterschiedliche Werte wie Null oder Eins zugewiesen haben, können wir nun klar diesem Pfad folgen, wenn wir einen terminalen Knoten erreichen wollen.

1. Beginn des Ablaufes: gegeben sei der Wurzelknoten:
    1. Wenn Knoten terminal ist: gespeicherten Pfad für den Buchstaben des derzeitigen Knotens vermerken, Funktionsaufruf beenden.
    2. Ansonsten, für jeden Kindknoten des Knotens:
        1. Den zugewiesenen Buchstaben/Wert aus den möglichen Encoding-Buchstaben von der Kante des Baumes, die zum Erreichen des Kindknotens genutzt wird, vermerken;

2. Rekursiv diesen Ablauf erneut aufrufen (zum äußersten Punkt 1 für den Kindknoten springen).

Offensichtlich geht dieses Verfahren nur auf, wenn der Baum zum Zeitpunkt der Decodierung auch vorliegt. Sollte dieser unbekannt sein, z.B. da er nicht gespeichert oder übermittelt wurde, ist eine Decodierung nicht möglich.

Unsere erste Aufgabe ist es nun, den Algorithmus so zu erweitern, dass er für das Konstruieren von n-ären Huffman-Bäumen nutzbar ist; somit wird es uns möglich, mehr als nur zwei verschiedene Perlenarten zu verwenden. Den Algorithmus zum Traversieren des Baumes müssen wir nicht anpassen. Die Konstruktion des Baumes verläuft geringfügig anders.

Statt immer die zwei seltensten Knoten zu verwenden, nehmen wir stattdessen die *n* seltensten, wobei n der Menge an gegebenen Buchstaben, die zur Codierung zur Verfügung stehen, entspricht.

Dieser Ansatz funktioniert zwar bereits, generiert aber einen suboptimalen Baum: Im Gegensatz zur konventionellen Huffman-Codierung ist unser Encoding nun nicht mehr das bestmögliche, da häufigere Knoten nicht mehr bei der optimalen Tiefe und Anordnung zum Wurzelknoten liegen. Lösen lässt sich dieses Problem durch die initiale Zusammenfassung von ggf. weniger als *n* Knoten zu einem übergeordneten Knoten, bevor man mit Schritt 3 des zuvor definierten Algorithmus zur Konstruktion eines Huffman-Baumes beginnt. Es ist zu ermitteln, wie viele Knoten initial zusammenzufassen sind.

Sei *N* diesmal die Anzahl an initialen Knoten vor Ausführung von Schritt 3. *D* sei die Größe des Encoding-Alphabets und somit gleichzeitig die N-arität des Baumes. Die Anzahl der zu kombinierenden Knoten im ersten Durchlauf sei *k*.

$$k = 2 + (N - 2) \bmod (D - 1)$$

Brechen wir diese Formel einmal herunter; zuerst fällt auf, dass 2 auf das Ergebnis von N – 2 modulo D – 1 addiert wird; dies ist notwendig, da – egal in welcher Ausführung – ein Huffman-Baum mindestens zwei Knoten in jedem Teilbaum zusammenfassen muss; es wäre sinnlos, einen Knoten mit lediglich einem weiteren, untergeordneten Knoten zu verknüpfen, da somit wertvolle Nachrichtenlänge unnötig verbraucht wird; der Knoten könnte schlichtweg gleich – ohne Umweg – ohne weitere Verknüpfung genutzt werden.

Somit ist ersichtlich, wieso wir mindestens zwei Kindknoten im initialen Teilbaum benötigen; die Modulo-Operation ermittelt nun, wie viele Kindknoten zusätzlich im initialen Schritt zu kombinieren sind, um einen optimal verteilten Huffman-Baum zu erreichen. Wir teilen somit N – 2 (zur Verfügung stehende Knoten) auf D – 1 (Zeichen im Encoding-Alphabet) auf und weisen den resultierenden Rest dem initialen Knoten zu; dies stellt sicher, dass – eben durch diese Modulo-Operation naturgemäß garantiert – nach der initialen Verteilung der unwichtigsten (und am tiefsten gelegenen) Knoten ab dann alle weiteren Knoten optimal bis zum dem Wurzelknoten am nächsten gelegenen Knoten hin verteilt werden können.

Implementierung eines Algorithmus zur Approximation eines optimalen Huffman-Baumes mit ungleichen Kosten der Encoding-Buchstaben

Den bisher beschriebenen Algorithmus gilt es nun insofern zu erweitern, dass dieser einen (möglichst) optimalen Huffman-Baum konstruiert, dabei aber eine weitere Einschränkung respektiert; die Buchstaben des Encoding-Alphabetes sind nun ungleich gewichtet, sie kosten unterschiedlich viel. Ziel ist es nicht mehr, die häufigsten Buchstaben des Ausgangstextes so nah wie möglich am Wurzelknoten zu platzieren, sondern den zu traversierenden Weg durch den Huffman-Baum für häufigere Buchstaben über tendenziell günstigere Kanten zu legen.

Dies führt eine neue Dimension an Komplexität ein; nicht länger ist der konventionelle Ansatz Huffmans gültig, um einen optimalen Baum zu konstruieren. In meinem Lösungsansatz habe ich dennoch versucht, den ursprünglichen Algorithmus Huffmans weitestgehend intakt zu halten und dabei umfangreicheres Augenmerk auf die Laufzeiteffizienz zu legen. Die zuvor genannte Formel zur Ermittlung der optimalen Anzahl

an Knoten bei der ersten Verknüpfung ist ebenfalls nicht mehr anwendbar; ersetzt wird sie durch das systematische Testen der Möglichkeiten zwischen $2 \leq k \leq D$.

Der Grundgedanke meines Algorithmus fußte ursprünglich darauf, bei jedem Schritt des Zusammenfügens – also zu jedem Moment, an welchem es n Knoten zu kombinieren gilt – den Kanten zum Knoten mit der höchsten Frequenz im Text jeweils den günstigsten Buchstaben im Encoding-Alphabet zuzuweisen.
Erweitert habe ich dieses Verfahren nun durch eine weitere Herangehensweise, um zu teure Buchstaben im Encoding-Alphabet seltener zu nutzen, dabei aber gleichzeitig durch die gezielte Nutzung der günstigeren innerhalb weiterer Teilbäume zu ersetzen.

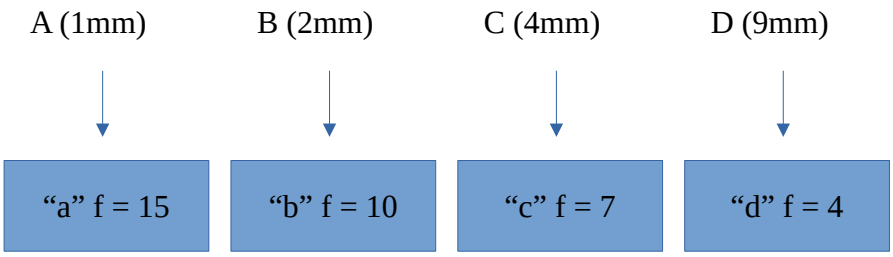
Wieder findet dieser Ansatz in der Funktion zum Zusammenfügen mehrerer Knoten statt und baut auf dem gerade genannten, ursprünglichen Ansatz auf. Der wichtige Unterschied ist hierbei, dass nicht einfach die Knoten bzw. Teilbäume mit der geringsten Frequenz die Kanten mit den teuersten Buchstaben im Encoding-Alphabet bekommen, sondern der Algorithmus schrittweise mögliche Kombinationen der – von dem Knoten der geringsten Frequenz bis zu dem der zweitteuersten – verfügbaren Knoten erprobt und diese rekursiv in Teilbäume zusammenfasst.

In den folgenden Schaubildern wird dieses Verfahren verbildlicht. Gegeben sei das folgende Encoding-Alphabet:

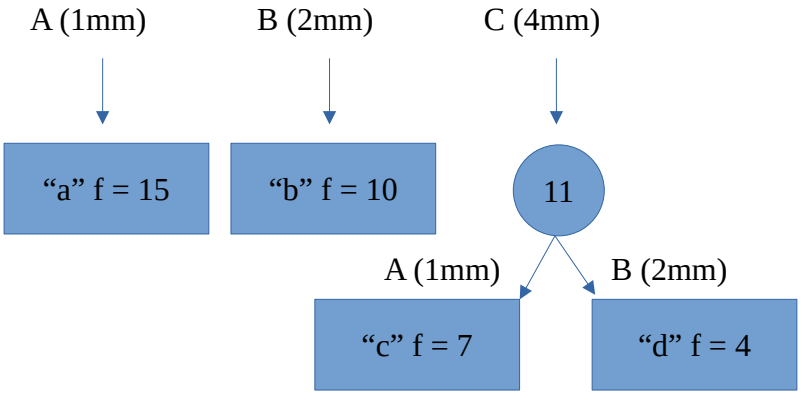
Encoding-Buchstabe	Kosten (z.B. in der Analogie in mm)
A	1
B	2
C	4
D	9

Logischerweise möchten wir den Buchstaben d eher meiden, soweit uns dies nicht durch die Teilbäume erhöhter Tiefe zum Nachteil wird. Wie dies ermittelt wird, wird später in dieser Dokumentation genauer betrachtet.

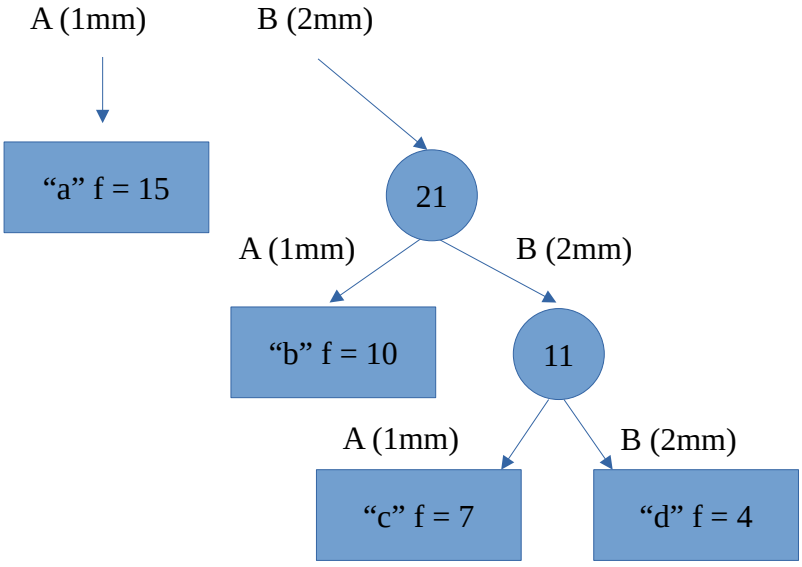
Mögliche Kombination 1 – nichts wird zusammengefasst, teuer:



Mögliche Kombination 2 – wir sparen uns die Verwendung eines teuren Ds mit 9mm, aber müssen jetzt natürlich immer einen Umweg über ein ebenfalls nicht ganz billiges C gehen:



Mögliche Kombination 3 – der rekursive Aspekt wird sichtbar. Wir rufen die Funktion zum Zusammenfügen in sich selbst auf, wodurch verschachtelte Strukturen wie diese entstehen können:



Dieser Ansatz ist recht zeiteffizient und sortiert tendenziell teurere Nutzungen der Buchstaben im Encoding-Alphabet in billigere um. Noch effektiver wird er, wenn pro Zusammenfassung mehrere Teilbäume erzeugt werden, nicht nur einer (rekursiv) für die teuersten.

Um zu entscheiden, welcher Grad an Gliederung optimal ist, müssen wir eine Bewertungsfunktion definieren, die die Kosteneffizienz unserer Anordnung bewertet.
Diese funktioniert folgendermaßen:

1. Für jeden zu kombinierenden Knoten, der durch die Merge-Funktion zusammengefasst werden soll:
    1. Multipliziere die Kosten des dem Knoten zugewiesenen Buchstabens aus dem Encoding-Alphabet mit der (summierten) Häufigkeit des Knotens und addiere das Ergebnis auf die Gesamtkosten;
    2. Sollte dies ein Endknoten sein, gehe zum nächsten Knoten.
    3. Ansonsten traversiere nun die Kinder dieses Knotens und führe dieselbe Operation rekursiv durch.

In Python ließe sich diese Kostenbewertung beispielsweise wie folgt implementieren (Ausschnitt aus der Implementierung):

```python
def get_cost_of_combination(
    nodes: "list[HuffmanNode]", alphabet: "dict[str, int]"
) -> int:
    """
    Ermittelt die Kosten einer Kombination innerhalb der Merge-Funktion.
    """

    cost = 0

    for node in nodes:
        cost += alphabet[node.assigned_code] * node.frequency

        if node.is_leaf():
            continue

        cost += get_cost_of_combination(node.children, alphabet)

    return cost
```

Das gesamte, optimierte Verfahren zur Codierung lässt sich nun also wie folgt beschreiben:

1. Die Häufigkeit jedes Buchstabens von dem aus der Datei eingelesenen Text auszählen und in einem Dictionary speichern.
2. Die Buchstaben in (noch unverbundene, terminale) Blattknoten umwandeln und nach Häufigkeit aufsteigend sortieren.
3. Mit der zuvor genannten Methode die bestmögliche Anzahl der initial zu kombinierenden Knoten ermitteln – einen Durchlauf für jede Option ausführen und die beste schlussendlich ausgeben (parallelisierbar). Für jede mögliche Anzahl:
    1. Solange es mehr als einen Knoten (der in diesem Fall der Wurzelknoten wäre) gibt:
        1. Liste an Knoten sortieren;
        2. Immer *n* (Anzahl an Buchstaben im Encoding-Alphabet) Knoten auswählen und mithilfe der Merge-Funktion zusammenfassen.
    2. Nach der Konstruktion des Baumes: den einzigen, übrigen Knoten nun vollständig rekursiv traversieren, um die Codetabelle zu erhalten. Dafür:
        1. Ein Dictionary anlegen, dass später für jeden Buchstaben im Text den zu gehenden Pfad durch den Baum, ergo, den präfixfreien Code parat hat.
        2. Dann: für jeden Kindknoten des derzeit traversierten Knotens (zu Beginn ist dies der Wurzelknoten):
            1. Dem der Kante, die den übergeordneten und den derzeitigen Kindknoten verbindet, zugewiesenen Buchstaben dem gegangenen Pfad hinzufügen.
            2. Überprüfen, ob der Knoten ein Blattknoten ist – wenn ja, den aufgezeichneten Pfad für den Buchstaben des Blattknotens in das Dictionary schreiben.
            3. Sonst: rekursiv für diesen Knoten den äußeren Schritt 2 (es ist die Rede von “*für jeden Kindknoten des derzeit* ...”) wieder aufrufen.
    3. Als Zusatzschritt kann zudem noch ein sogenanntes „Re-Mapping“ des Encodings weitere Qualitätsverbesserungen bringen. Dieses ist extrem einfach, hat eine lineare Zeitkomplexität und ist deshalb auch bei mir implementiert:
        1. Buchstabenhäufigkeiten werden aufsteigend sortiert,
        2. Encoding-Alphabet wird nach Kosten aufsteigend sortiert,

3. Die günstigsten Einträge im Encoding werden nach und nach den häufigsten Buchstaben zugewiesen (und umgekehrt).

Letztendlich kann dann das Dictionary als Tabelle ausgegeben werden.

Nun zur Merge-Funktion – ihre Aufgabe ist das optimierte Zusammenfügen mehrerer Knoten, um einen Huffman-Baum bei ungleichen Kosten der Buchstaben im Encoding-Alphabet zu approximieren.

1. Überprüfe, ob es nur einen Knoten zu verknüpfen gibt. In diesem Falle, tue nichts – einen einzigen Knoten kann man nicht zusammenfassen.
2. Sortiere das Encoding-Alphabet von günstig nach teuer.
3. Wenn mehr als zwei Knoten zu kombinieren sind:
    1. Wähle systematisch – vom teuersten Knoten aus, immer weiter in Richtung günstigere Knoten – mehr und mehr Knoten, die potenziell zusammengefasst werden könnten, aus. Für jede Auswahl:
        1. Rufe diese Merge-Funktion rekursiv auf diese Menge Knoten auf.
        2. Weise dieser möglichen Permutation unserer Anordnung hypothetische Buchstaben an den Kanten zu.
        3. Ermittle rekursiv, was für Kosten diese Kombination dann hätte (*siehe vorher genannte Funktion zur Kostenbewertung*).
        4. Wenn dies die besten Kosten sind, die wir bisher hatten: Permutation merken.
    2. Ansonsten, wähle nur die zwei zu kombinierenden Knoten aus. Hier gibt es nichts zu optimieren, da uns bei lediglich zwei Knoten die Möglichkeiten fehlen. So oder so müssen wir einfach die zwei günstigsten Buchstaben des Encoding-Alphabets zuweisen.
4. Danach von der besten gefundenen Permutation (bzw. einfach nur den zwei Knoten) die Äste entsprechend “beschriften”, indem die häufigsten Knoten die günstigsten Buchstaben bekommen.
5. Neues Knoten-Objekt mit den Knoten der Permutation als Kinder erstellen, dabei deren Frequenz summieren und als Frequenz des neuen Knotens verwenden.

6. Die alten Knoten, die ursprünglich kombiniert werden sollten, nun aus der originalen Liste löschen und durch den zusammengefügten Knoten ersetzen.

Dies fasst den Algorithmus zur Erstellung einer optimalen Codetabelle zusammen.

Relevante Ausschnitte der Implementierung

Folgend gelistet sind die relevantesten Teile der Implementierung des oben genannten Algorithmus. Vorab sei gesagt, dass die gesamte Datei – nicht nur die Dokumentation – sehr umfassende Erklärungen und Kommentare enthält. Etwaige Fragen, die beim Lesen aufkommen, sollten von den Docstrings und Kommentaren beantwortet werden könnten.

1. Code zum Konstruieren des Huffman-Baumes:

```python
def construct_tree(
    occurences: "dict[str, int]",
    alphabet: "dict[str, int]",
    n_arity: int,
    override_nodes_in_first_merge: int = -1,
) -> HuffmanNode:
    """
    Konstruiert den Huffman-Baum aus drei Informationen: der Anzahl und
    Verteilung der Buchstaben im Text, dem gewichteten Encoding-Alphabet und
    der n-Arität, also der Anzahl an möglichen Buchstaben im Encoding-Alphabet
    und somit der maximalen Anzahl an Kanten, die zu Kindknoten führen, eines
    jeden Knotens.

    Der Override für die Menge an Knoten beim ersten Zusammenfügen kann auf
    -1 gesetzt werden, um diesen nicht zu nutzen.
    """

    nodes = [
        HuffmanNode(freq=v, children=[], char=k) for k, v in occurences.items()
    ]

    def sort_nodes_list(nodes: "list[HuffmanNode]") -> None:
        nodes.sort(key=lambda el: el.frequency)

    sort_nodes_list(nodes)

    if override_nodes_in_first_merge != -1:
        to_pack: int = override_nodes_in_first_merge
    else:
        # Initialer Run - wir führen die erste Zusammenfügung separat aus
        # (siehe Docstring in how_many_in_first_node für Erklärung)
        to_pack: int = how_many_in_first_node(len(occurences), len(alphabet))

    merge_nodes(nodes, to_pack, alphabet)

    # Danach geht es weiter mit dem Hauptvorgang: Wir verknüpfen unsere Nodes
    # so lange, bis nur noch ein Knoten übrig ist. Dieser ist dann die
    # "Wurzel".
    while len(nodes) > 1:
        sort_nodes_list(nodes)
        merge_nodes(nodes, n_arity, alphabet)

    return nodes[0]
```

2. Code zum Zusammenfügen der Knoten (Merge-Funktion), Kommentare und Docstrings der Lesbarkeit in diesem Dokument halber gekürzt:

```python
def merge_nodes(
    nodes: list[HuffmanNode], n: int, alphabet: "dict[str, int]"
) -> None:
    if n == 1:
        return

    nodes_to_merge: "list[HuffmanNode]" = nodes[:n]
    alphabet_letters_to_assign: "OrderedDict[str, int]" = sorted_dict(alphabet)
    alphabet_letters: "list[str]" = list(alphabet_letters_to_assign.keys())

    if n > 2:
        best_permutation = nodes_to_merge
        best_cost = inf

        for to_merge in range(1, min(len(alphabet), len(nodes_to_merge))):
            cost: int = 0
            permutation = list(reversed(deepcopy(nodes_to_merge)))

            if to_merge > 1:
                merge_nodes(permutation, to_merge, alphabet)

            permutation = list(reversed(permutation))

            for index, el in enumerate(
                sorted(permutation, key=lambda el: el.frequency, reverse=True)
            ):
                el.assigned_code = alphabet_letters[index]

            cost = get_cost_of_combination(permutation, alphabet)
            if cost < best_cost:
                best_cost = cost
                best_permutation = permutation

        nodes_to_merge = best_permutation

    for index, el in enumerate(
        sorted(nodes_to_merge, key=lambda el: el.frequency, reverse=True)
    ):
        el.assigned_code = alphabet_letters[index]

    new_node = HuffmanNode(
        "", nodes_to_merge, sum(node.frequency for node in nodes_to_merge)
    )

    del nodes[:n]
    nodes.insert(0, new_node)
```

3. Code zur Erstellung der Codetabelle aus dem Huffman-Baum (teils Docstrings gekürzt):

```python
def generate_encoding(root_node: HuffmanNode) -> "dict[str, str]":
    """
    Simple Methode, die schlichtweg jeden möglichen Pfad durch den Baum
    traversiert und aufzeichnet, welcher Weg für welchen Buchstaben zu gehen
    ist. Daraus wird unser präfixfreies Encoding generiert.
    """

    encoding: "dict[str, str]" = {}

    def traverse(node: HuffmanNode, cur_code: str):
        """
        Geht rekursiv durch den Baum und behält den gegangenen Weg im Auge.
        """

        if node.is_leaf():
            # Aha, wir sind am Ende angekommen.
            encoding[node.char] = cur_code
            return

        for child in node.children:
            # Wenn wir noch nicht am Ende eines Pfades angekommen sind, schauen
            # wir alle untergeordneten Knoten durch.
            traverse(child, cur_code + child.assigned_code)

    # Ursprungsaufruf zum Beginn
    traverse(root_node, "")

    return encoding
```

```python
def remap_encoding(
    encoding: "dict[str, str]",
    occurences: "dict[str, int]",
    alphabet: "dict[str, int]",
) -> "dict[str, str]":
    sorted_occurences = sorted_dict(occurences)
    costs: "list[tuple[str, int]]" = sorted(
        [
            (v, encode_text(k, encoding, alphabet)[1])
            for k, v in encoding.items()
        ],
        key=lambda el: el[1],
    )

    new_dict: "dict[str, str]" = {}

    for k in sorted_occurences:
        new_dict[k] = costs.pop()[0]

    return new_dict
```

4. Code zum Ermitteln der Menge an zu kombinierenden Knoten in dem Ersten Durchlauf:

```python
def how_many_in_first_node(n: int, d: int) -> int:
    """
    Ermittelt die Anzahl an Leaf Nodes, die in der ersten Zusammenfassung bei
    der Konstruktion eines n-ären Huffman-Baumes vorhanden sein sollen.

    Ohne diese Berechnung riskieren wir einen suboptimalen Baumaufbau.
    """

    # n = wie viele verschiedene Buchstaben im Text wir haben
    # d = wie viele Encoding-Buchstaben wir haben (N-ärität)
    return 2 + (n - 2) % (d - 1)
```

5. Der Code der so häufig verwendeten Klasse `HuffmanNode` (*wieder ohne Kommentare und Docstrings, volle Version in der Implementierungsdatei*):

```
class HuffmanNode:
    frequency = 0
    children = []
    char = ""
    assigned_code = ""

    def is_leaf(self) -> bool:
        return self.char != "" and not self.children

    def __init__(
        self,
        char: str,
        children: "list[HuffmanNode]",
        freq: int,
        assigned_code: str = "",
    ) -> None:
        self.char: str = char
        self.children: "list[HuffmanNode]" = children
        self.frequency: int = freq
        self.assigned_code: str = assigned_code

    def __repr__(self) -> str:
        return f'<{self.assigned_code + "=" if self.assigned_code else ""}N \
{self.char + " " if self.char else ""}f={self.frequency} c={self.children}>'
```

Details zur Implementierung

Die Implementierung des oben erläuterten Algorithmus wurde in der Programmiersprache Python realisiert. Sie ist in Form einer Kommandozeilenanwendung nutzbar.

Es gibt mehrere verschiedene Command-Line-Flags, die die Funktionalität des Programms mit Zusatzfunktionen verbessern. Diese lauten:

Flag	**Funktionalität**
`--tree`	Gibt neben der Codetabelle zudem eine Visualisierung des generierten Codebaumes aus, mit der man die Ausgabe besser inspizieren und nachvollziehen kann.
`--verbose`	Gibt noch mehr Informationen aus; darunter der codierte Text und die individuellen Codierungskosten pro Buchstabe.
`--time`	Führt Zeitmessungen durch und gibt nach Ausführung des Programms Zusatzinformationen zur Geschwindigkeit aus.
`--compact`	Sorgt für eine kompaktere Ausgabe (hier verwendet, um die Programmausgaben in der Dokumentation kürzer zu halten)
`--help`	Gibt Informationen zur korrekten Programmnutzung aus.

Ein Beispiel zur Ausführung des Programms im Terminal wäre:

```
$ pypy ./huffman.py ./Beispiele/schmuck0.txt --tree --time --verbose
```

Die resultierende Codetabelle (und auch die Visualisierung des Huffman-Baums) wird mittels Box Drawing Characters[1] anschaulicher im Terminal dargestellt[2]:

```
Die optimierte Tabelle lautet:
┌─┬──┐
│G│ca│
│F│cb│
│E│aa│
│B│ab│
│A│ac│
│C│ba│
│H│bb│
│D│bc│
└─┴──┘
Textlänge: 8 · Encodete Länge: 16 Perlen · Gesamtkosten (∑): 30mm
Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!
```

Weitere Eigenschaften der Implementierung:

- prüft von selbst, ob die generierte Codetabelle auch wirklich präfixfrei ist,
- ist umfassend dokumentiert und kommentiert,
- nutzt keinerlei Bibliotheken von Dritten,
- ist sehr streng typisiert; weder Linter-Fehler noch Linter-Warnungen,
- sollte dem PEP8-Standard vollständig entsprechen und konform sein,
- läuft problemlos in verschiedenen Interpretern, darunter CPython und PyPy,
- kann unter Anwendung von JIT-Compilation sehr hohe Codiergeschwindigkeiten erreichen,
- folgt den UNIX-Paradigmen und gibt korrekte Exit-Codes an den ausführenden Elternprozess zurück,
- baut Bäume und deren Encoding-Tabellen zu Texten mit mehreren MiB/s Encoding-Geschwindigkeit auf (in meinen Messungen bis über 10 MiB/s – für Python passabel).

1 https://de.wikipedia.org/wiki/Unicodeblock_Rahmenzeichnung

2 Auf Windows-Kommandozeilen können einige Unicode-Symbole womöglich nicht korrekt angezeigt werden, da die falsche Codepage ausgewählt ist. Dies ist kein Fehler im Programm und rein betriebssystemabhängig.

Ausgaben des Programms zu den Beispieleingaben von der Website des BWINF

Folgend sind die Ausgaben des Programms zu den Beispieleingaben von der Website des BWINF sichtbar. Berechnet wurden sie mit folgendem Setup:

Betriebssystem	Ubuntu 22.04, kernel 6.8.0-57-generic
Interpreter	PyPy 3.11 v7.3.19
CPU	AMD Ryzen 5 6600U 6 Kerne @ 4.5 GHz
Arbeitsspeicher	16 GB DDR4

```
--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck0.txt --- ---

│ "C" = abbab │ "H" = abbaa │ "D" = abbb │ "O" = abab │ "L" = abaa │ "R" =
bbab │ "M" = bbaa │ "S" = aab │ "I" = aaa │ "N" = bab │ "E" = baa │ " " =
bbb │

Textlänge: 33 · Encodete Länge: 113 Perlen · Gesamtkosten (∑): 113mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck00.txt --- ---

│ "E" = aaabc │ "k" = aaabb │ "f" = aaaba │ "W" = aaaac │ "Z" = aaaab │ "P"
= aaaaa │ "o" = bcaab │ "?" = bcaaa │ "A" = aaac │ "b" = acbc │ "w" = acbb
│ "m" = acba │ "G" = bcac │ "g" = bcab │ "d" = aac │ "D" = aab │ "u" = aca
│ "a" = acc │ "r" = bcc │ "l" = bcb │ "c" = cbc │ "s" = cbb │ "h" = cba │
"n" = ab │ "t" = ba │ "i" = bb │ "e" = ca │ " " = cc │

Textlänge: 141 · Encodete Länge: 372 Perlen · Gesamtkosten (∑): 372mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck1.txt --- ---

│ "W" = abcc │ "N" = abcb │ "F" = abca │ "h" = abbc │ "Ü" = abac │ "D" =
aaac │ "u" = abbb │ "d" = abba │ "o" = abab │ "m" = abaa │ "a" = aac │ "k"
= aaab │ "B" = aaaa │ "I" = bbc │ "s" = bac │ "f" = cc │ """ = ac │ "b" =
aab │ "r" = bc │ "i" = bbb │ "n" = bba │ "w" = bab │ " " = baa │ "t" = ca │
"e" = cb │

Textlänge: 56 · Encodete Länge: 159 Perlen · Gesamtkosten (∑): 195mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck01.txt --- ---

│ "K" = aaad │ "R" = aaac │ "M" = aaab │ "ß" = aaaa │ "H" = aaae │ "N" =
baab │ "W" = baaa │ "…" = baae │ "B" = baad │ "ä" = baac │ "G" = bace │ "O"
= bacd │ "V" = bacc │ "ö" = bacb │ "z" = baca │ "F" = aab │ "f" = aac │ "S"
```

```
= aad │ "v" = aae │ "I" = aca │ "p" = acc │ "ü" = acb │ "w" = ace │ "D" =
acd │ "b" = bab │ "E" = bad │ "," = bae │ "d" = dad │ "k" = dac │ "." = dab
│ "u" = daa │ "m" = dae │ "o" = ab │ "g" = ad │ "c" = ae │ "h" = bb │ "l" =
be │ "a" = bd │ "t" = bc │ "s" = db │ "i" = dc │ "r" = de │ "n" = dd │ "e"
= c │ " " = e │

Textlänge: 566 · Encodete Länge: 1150 Perlen · Gesamtkosten (∑): 1150mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck2.txt --- ---

│ " " = bbbb │ "b" = bbba │ "c" = bbab │ "d" = babb │ "e" = bbaa │ "f" =
baba │ "g" = baab │ "h" = baaa │ "a" = a │

Textlänge: 41 · Encodete Länge: 65 Perlen · Gesamtkosten (∑): 145mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck3.txt --- ---

│ "d" = abac │ "e" = abab │ "f" = abc │ "g" = abaa │ "h" = abb │ " " = ac │
"a" = c │ "b" = aa │ "c" = b │

Textlänge: 110 · Encodete Länge: 160 Perlen · Gesamtkosten (∑): 279mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck4.txt --- ---

│ "a" = abbb │ "b" = babb │ "c" = bbb │ "d" = abba │ "e" = abab │ "f" =
aabb │ "g" = baba │ "h" = baab │ "i" = bba │ "j" = abaa │ "k" = aaba │ "l"
= aaab │ "m" = baaa │ "n" = aaaa │

Textlänge: 14 · Encodete Länge: 54 Perlen · Gesamtkosten (∑): 154mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck5.txt --- ---

│ "S" = gce │ "-" = gcac │ "A" = gcd │ "H" = gcab │ "1" = gcaa │ "9" = gcc
│ "5" = ge │ "2" = gac │ "z" = gcb │ "G" = gd │ "’" = bbacd │ "j" = gab │
";" = gaa │ "T" = bbacab │ "F" = bbacaa │ "w" = bbacc │ "v" = gb │ "x" =
bbacb │ "q" = bbe │ "," = ae │ "." = bbd │ "g" = be │ "y" = f │ "b" = aac │
"p" = ad │ "f" = bac │ "u" = bbab │ "h" = bbaa │ "m" = bbc │ "l" = bd │ "d"
= e │ "c" = aab │ "r" = aaa │ "n" = ac │ "a" = bab │ "s" = baa │ "o" = bbb
│ "i" = bc │ "t" = d │ "e" = ab │ " " = c │

Textlänge: 1012 · Encodete Länge: 2275 Perlen · Gesamtkosten (∑): 3229mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck6.txt --- ---

│ "古" = ccc │ "英" = ccb │ "雄" = cbc │ "未" = bcc │ "遇" = aabc │ "時" =
abac │ "都" = acc │ "無" = acab │ "大" = cca │ "志" = cbb │ "非" = cac │
"止" = bcb │ "鄧" = bbc │ "禹" = aabb │ "希" = aaac │ "學" = abab │ "、" =
abc │ "武" = acb │ "望" = acaa │ "督" = cba │ "郵" = cab │ "也" = bca │ "晉"
= bbb │ "公" = bac │ "妻" = aac │ "不" = aaba │ "肯" = aaab │ "去" = abaa │
```

"齊" = abb │ "文" = caa │ "馬" = bba │ "。" = bab │ "有" = aaaa │ ", " = baa │

Textlänge: 40 · Encodete Länge: 132 Perlen · Gesamtkosten (∑): 237mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck7.txt --- ---

│ "Ö" = achhj │ "x" = achhi │ "y" = achhh │ "C" = achhb │ "7" = achha │ "9" = achhf │ "Q" = achhe │ "’" = achhd │ "q" = achhc │ "Ä" = achhg │ "8" = dgcj │ "0" = achac │ "6" = achab │ "2" = achaa │ "!" = agj │ "5" = dgci │ "?" = dgj │ "3" = achb │ "4" = achd │ "1" = achc │ "P" = achf │ "„" = ache │ "“" = achg │ "[" = agi │ "]" = aj │ "U" = dgch │ "O" = dgi │ ":" = dj │ "_" = acac │ "(" = acab │ ")" = acaa │ ";" = agh │ "-" = ai │ "J" = dgca │ "j" = dgcb │ "Z" = dgcc │ "T" = dgcd │ "G" = dgcg │ "R" = dgcf │ "L" = dgce │ "I" = dgh │ "H" = di │ "A" = j │ "M" = acc │ "B" = acb │ "K" = acd │ "N" = ace │ "W" = acf │ "V" = acg │ "E" = aga │ "D" = agb │ "ß" = agc │ "p" = agd │ "F" = agf │ "v" = age │ "ö" = agg │ "Ü" = ah │ "ä" = dga │ "S" = dgb │ "z" = dgd │ "k" = dge │ "." = dgf │ "w" = dgg │ "f" = dh │ "," = i │ "b" = aa │ "o" = ab │ "m" = ad │ "g" = ae │ "c" = af │ "l" = da │ "u" = db │ "t" = dc │ "h" = dd │ "d" = de │ "a" = df │ "s" = h │ "r" = b │ "i" = c │ "n" = e │ "e" = f │ " " = g │

Textlänge: 82579 · Encodete Länge: 128353 Perlen · Gesamtkosten (∑): 137043mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck8.txt --- ---

│ "古" = ceee │ "英" = beee │ "雄" = ceed │ "遇" = ceec │ "止" = cede │ "鄧" = cece │ "希" = cdee │ "學" = ccee │ "望" = beed │ "督" = beec │ "郵" = bede │ "晉" = bece │ "妻" = bdee │ "肯" = bcee │ "去" = ceeb │ "齊" = ceea │ "貧" = cedd │ "訟" = cedc │ "逋" = cecd │ "租" = cecc │ "於" = cebe │ "奇" = ceae │ "歸" = cded │ "謂" = cdec │ "寧" = cdde │ "耶" = cdce │ "窺" = cced │ "盼" = ccec │ "榮" = ccde │ "蘄" = ccce │ "小" = cbee │ "卒" = caee │ "士" = abee │ "封" = acde │ "怒" = acce │ "侮" = beeb │ "己" = beea │ "奮" = bedd │ "拳" = bedc │ "毆" = becd │ "般" = becc │ "鄂" = bebe │ "西" = beae │ "辛" = bded │ "丑" = bdec │ "元" = bdde │ "攬" = bdce │ "鏡" = bced │ "老" = bcec │ "門" = bcde │ "草" = bcce │ "生" = cedb │ "詠" = ceda │ "懷" = cecb │ "猶" = ceca │ "如" = cebd │ "到" = cebc │ "可" = cead │ "郎" = ceac │ "玩" = cdeb │ "詞" = cdea │ "若" = cddd │ "料" = cddc │ "入" = cdcd │ "及" = cdcc │ "省" = cdbe │ "經" = cdae │ "略" = cceb │ "金" = ccea │ "丞" = ccdd │ "席" = ccdc │ "心" = cccd │ "酬" = cccc │ "恩" = ccbe │ "客" = ccae │ "屈" = cbed │ "指" = cbec │ "世" = cbde │ "登" = cbce │ "甲" = caed │ "秀" = caec │ "樓" = cade │ "絕" = cace │ "句" = dace │ "炊" = dee │ "煙" = aace │ "卓" = abed │ "午" = abec │ "散" = abde │ "輕" = abce │ "絲" = acdd │ "萬" = acdc │ "家" = accd │ "飯" = accc │ "熟" = acbe │ "訊" = acae │ "招" = adae │ "火" = bedb │ "斜" = beda │ "陽" = becb │ "樹" = beca │ "鄉" = bebd │ "祠" = bebc │ "居" = bead │ "然" = beac │ "侯" = bdeb │ "命" = bdea │ "氣" = bddd │ "象" = bddc │ "迥" = bdcd

```
│ "異" = bdcc │ "張" = bdbe │ "桐" = bdae │ "城" = bceb │ "則" = bcea │ "翰"
= bcdd │ "至" = bcdc │ "首" = bccd │ "最" = bccc │ "清" = bcbe │ "妙" = bcae
│ "柳" = cebb │ "陰" = ceba │ "春" = ceab │ "曲" = ceaa │ "暮" = cddb │ "山"
= cdda │ "葉" = cdcb │ "底" = cdca │ "雙" = cdbd │ "蝴" = cdbc │ "蝶" = cdad
│ "先" = cdac │ "臨" = ccdb │ "種" = ccda │ "化" = cccb │ "兩" = ccca │ "扈"
= ccbd │ "蹕" = ccbc │ "憐" = ccad │ "龍" = ccac │ "鐘" = cbeb │ "叟" = cbea
│ "騎" = cbdd │ "踏" = cbdc │ "冰" = cbcd │ "星" = cbcc │ "和" = cbbe │ "皇"
= cbae │ "〈" = caeb │ "箏" = caea │ "九" = cadd │ "霄" = cadc │ "近" = cacd
│ "增" = cacc │ "華" = cabe │ "色" = caae │ "野" = dacd │ "仗" = dacc │ "寶"
= dabe │ "押" = daae │ "字" = dce │ "韻" = ded │ "寄" = dec │ "托" = dde │
"遙" = eae │ "二" = ebe │ "楊" = aacd │ "誠" = aacc │ "齋" = abeb │ "從" =
abea │ "分" = abdd │ "低" = abdc │ "拙" = abcd │ "空" = abcc │ "架" = abbe │
"子" = abae │ "腔" = acdb │ "口" = acda │ "易" = accb │ "描" = acca │ "專" =
acbd │ "寫" = acbc │ "靈" = acad │ "辦" = acac │ "餘" = ace │ "愛" = adad │
"須" = adac │ "半" = ade │ "勞" = bebb │ "思" = beba │ "婦" = beab │ "率" =
beaa │ "事" = bddb │ "今" = bdda │ "能" = bdcb │ "範" = bdca │ "圍" = bdbd │
"否" = bdbc │ "況" = bdad │ "皋" = bdac │ "歌" = bcdb │ "國" = bcda │ "雅" =
bccb │ "頌" = bcca │ "豈" = bcbd │ "定" = bcbc │ "哉" = bcad │ "許" = bcac │
"渾" = cdbb │ "似" = cdba │ "成" = cdab │ "仙" = cdaa │ "里" = ccbb │ "莫" =
ccba │ "浪" = ccab │ "無" = ccaa │ "大" = cbdb │ "志" = cbda │ "非" = cbcb │
"禹" = cbca │ "文" = cbbd │ "光" = cbbc │ "與" = cbad │ "李" = cbac │ "通" =
cadb │ "尤" = cada │ "目" = cacb │ "曰" = caca │ "君" = cabd │ "得" = cabc │
"韓" = caad │ "王" = caac │ "後" = dae │ "見" = dacb │ "解" = daca │ "林" =
dabd │ "將" = dabc │ "開" = daad │ "看" = daac │ "來" = dbe │ "四" = dcd │
"此" = dcc │ "便" = deb │ "年" = dea │ "已" = ddd │ "中" = ddc │ "出" = ead
│ "七" = eac │ "上" = ebd │ "問" = ebc │ "濟" = ee │ "才" = aae │ "何" =
aacb │ "滿" = aaca │ "自" = abdb │ "水" = abda │ "外" = abcb │ "多" = abca │
"枝" = abbd │ "繩" = abbc │ "深" = abad │ " " = abac │ "好" = acbb │ "談" =
acba │ "趣" = acab │ "三" = acaa │ "篇" = adab │ "同" = adaa │ "乎" = add │
"吟" = adc │ "未" = bae │ "都" = bdbb │ "、" = bdba │ "馬" = bdab │ "嚴" =
bdaa │ "而" = bcbb │ "意" = bcba │ "以" = bcab │ "言" = bcaa │ "日" = cbbb │
"十" = cbba │ "百" = cbab │ "皆" = cbaa │ "作" = cabb │ "者" = caba │ "花" =
caab │ "天" = caaa │ "調" = dad │ "性" = dabb │ "情" = daba │ "律" = daab │
"骨" = daaa │ "也" = dbd │ "公" = dbc │ "是" = dcb │ "知" = dca │ "在" = ddb
│ "誰" = dda │ "; " = eab │ "武" = eaa │ "相" = ebb │ "人" = eba │ "詩" = ed
│ "風" = ec │ "時" = aad │ "其" = abbb │ "一" = abba │ "有" = abab │ "為" =
abaa │ "云" = adb │ "不" = ae │ "之" = bad │ "格" = bac │ ": " = dbb │ "《"
= dba │ "》" = aab │ "「" = aaa │ "」" = bab │ "。" = baa │ ", " = bb │
```


Textlänge: 633 · Encodete Länge: 2216 Perlen · Gesamtkosten (∑): 3427mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- --- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck9.txt --- ---

```
│ "英" = cbdddd │ "国" = cbdddc │ "測" = cbddcd │ "制" = cbdcdd │ "作" =
cbcddd │ "単" = abdddd │ "敵" = cacddd │ "艦" = cbdddb │ "戦" = cbddcc │
"六" = cbddbd │ "紀" = cbdcdc │ "攻" = cbdccd │ "迎" = cbdbdd │ "緒" =
cbcddc │ "歴" = cbcdcd │ "史" = cbccdd │ "輝" = cbbddd │ "残" = abdddc │
"域" = abddcd │ "沈" = abdcdd │ "泥" = cacddc │ "ふ" = cacdcd │ "州" =
cbddda │ "貿" = cbddcb │ "易" = cbddbc │ "探" = cbddad │ "へ" = cbdcdb │
"縦" = cbdccc │ "縮" = cbdcbd │ "変" = cbdbdc │ "貌" = cbdbcd │ "類" =
cbdadd │ "計" = cbcddb │ "塩" = cbcdcc │ "沢" = cbcdbd │ "埋" = cbccdc │
"償" = cbcccd │ "真" = cbcbdd │ "路" = cbbddc │ "造" = cbbdcd │ "低" =
cbbcdd │ "峡" = cbaddd │ "プ" = abdddb │ "産" = abddcc │ "抵" = abddbd │
"抗" = abdcdc │ "渡" = abdccd │ "東" = abdbdd │ "潮" = acaddd │ "共" =
cacddb │ "忘" = cacdcc │ "昔" = cacdbd │ "拘" = cbddca │ "束" = cbddbb │
"断" = cbddac │ "切" = cbdcda │ "暴" = cbdccb │ "階" = cbdcbc │ "眺" =
cbdcad │ "誰" = cbdbdb │ "再" = cbdbcc │ "移" = cbdbbd │ "浸" = cbdadc │
"湖" = cbdacd │ "界" = cbcdda │ "幅" = cbcdcb │ "ザ" = cbcdbc │ "幹" =
cbcdad │ "捨" = cbccdb │ "七" = cbcccc │ "長" = cbccbd │ "鄙" = cbcbdc │
"往" = cbcbcd │ "選" = cbcadd │ "議" = cbbddb │ "ゼ" = cbbdcc │ "体" =
cbbdbd │ "表" = cbbcdc │ "正" = cbbccd │ "組" = cbbbdd │ "織" = cbaddc │
"改" = cbadcd │ "善" = cbacdd │ "必" = ccabdd │ "要" = cdcabd │ "設" =
abddda │ "性" = abddcb │ "去" = abddbc │ "比" = abddad │ "粗" = abdcdb │
"末" = abdccc │ "託" = abdcbd │ "当" = abdbdc │ "珍" = abdbcd │ "消" =
acaddc │ "毎" = acadcd │ "激" = acacdd │ "ロ" = bcdcd │ "旅" = caddd │ "費"
= cacdda │ "節" = cacdcb │ "約" = cacdbc │ "換" = cacdad │ "威" = caccbd │
"苦" = cbddba │ "労" = cbddab │ "終" = cbdcca │ "養" = cbdcbb │ "院" =
cbdcac │ "配" = cbdbda │ "属" = cbdbcb │ "ズ" = cbdbbc │ "族" = cbdbad │
"喜" = cbdadb │ "好" = cbdacc │ "客" = cbdabd │ "余" = cbcdca │ "裕" =
cbcdbb │ "ぷ" = cbcdac │ "藁" = cbccda │ "突" = cbcccb │ "間" = cbccbc │
"耐" = cbccad │ "完" = cbcbdb │ "駆" = cbcbcc │ "角" = cbcbbd │ "聳" =
cbcadc │ "セ" = cbcacd │ "嘆" = cbbdda │ "声" = cbbdcb │ "抑" = cbbdbc │
"篠" = cbbdad │ "追" = cbbcdb │ "払" = cbbccc │ "緑" = cbbcbd │ "幾" =
cbbbdc │ "デ" = cbbbcd │ "ィ" = cbbadd │ "ゴ" = cbaddb │ "待" = cbadcc │
"沛" = cbadbd │ "然" = cbacdc │ "砕" = cbaccd │ "霧" = cbabdd │ "昇" =
ccabdc │ "蒸" = ccabcd │ "ェ" = ccaadd │ "被" = cdcabc │ "台" = cdcaad │
"紗" = aaaddd │ "幕" = abddca │ "郭" = abddbb │ "浮" = abddac │ "想" =
abdcda │ "像" = abdccb │ "描" = abdcbc │ "姿" = abdcad │ "遥" = abdbdb │
"秘" = abdbcc │ "的" = abdbbd │ "荘" = acaddb │ "門" = acadcc │ "停" =
acadbd │ "抜" = acacdc │ "板" = acaccd │ "御" = accacd │ "開" = adbdd │ "届"
= bcdcc │ "帽" = bcdbd │ "子" = caddc │ "襟" = cadcd │ "飛" = cacdca │ "墓"
= cacdbb │ "急" = cacdac │ "服" = caccbc │ "撥" = caccad │ "扉" = cbddaa │
"戸" = cbdcba │ "革" = cbdcab │ "帳" = cbdbca │ "《" = cbdbbb │ "》" =
cbdbac │ "押" = cbdada │ "雲" = cbdacb │ "閉" = cbdabc │ "薄" = cbdaad │
"歌" = cbcdba │ "席" = cbcdab │ "男" = cbccca │ "返" = cbccbb │ "領" =
cbccac │ "袖" = cbcbda │ "首" = cbcbcb │ "巻" = cbcbbc │ "挨" = cbcbad │
"拶" = cbcadb │ "彩" = cbcacc │ "使" = cbcabd │ "同" = cbbdca │ "己" =
```

```
cbbdbb │ "申" = cbbdac │ "聞" = cbbcda │ "別" = cbbccb │ "付" = cbbcbc │
"泊" = cbbcad │ "業" = cbbbdb │ "誇" = cbbbcc │ "回" = cbbbbd │ "短" =
cbbadc │ "敬" = cbbacd │ "調" = cbadda │ "相" = cbadcb │ "扱" = cbadbc │
"貶" = cbadad │ "疑" = cbacdb │ "問" = cbaccc │ "ャ" = cbacbd │ "礼" =
cbabdc │ "拝" = cbabcd │ "演" = cbaadd │ "ニ" = ccabdb │ "優" = ccabcc │
"秀" = ccabbd │ "売" = ccaadc │ "管" = ccaacd │ "理" = ccbcd │ "伝" = cdcabb
│ "受" = cdcaac │ "肩" = cdcad │ "態" = aaaddc │ "侮" = aaadcd │ "蔑" =
aaacdd │ "万" = abddba │ "逆" = abddab │ "充" = abdcca │ "認" = abdcbb │
"識" = abdcac │ "恭" = abdbda │ "揖" = abdbcb │ "丁" = abdbbc │ "寧" =
abdbad │ "謙" = acadda │ "虚" = acadcb │ "惜" = acadbc │ "護" = acadad │
"務" = acacdb │ "料" = acaccc │ "機" = acacbd │ "嫌" = acabcd │ "徴" =
accacc │ "摘" = accabd │ "折" = adbdc │ "快" = adbcd │ "褒" = badcd │ "逃" =
bcdcb │ "観" = bcdbc │ "念" = bcdad │ "傘" = bdcad │ "独" = caddb │ "雰" =
cadcc │ "囲" = cadbd │ "醸" = cacdba │ "冷" = cacdab │ "漏" = caccd │ "雄" =
caccbb │ "弁" = caccac │ "案" = cbdcaa │ "千" = cbdbba │ "百" = cbdbab │
"五" = cbdaca │ "経" = cbdabb │ "験" = cbdaac │ "脇" = cbcdaa │ "尋" =
cbccba │ "導" = cbccab │ "夫" = cbcbca │ "答" = cbcbbb │ "眉" = cbcbac │
"毛" = cbcada │ "崇" = cbcacb │ "高" = cbcabc │ "簡" = cbcaad │ "ケ" =
cbbdba │ "吟" = cbbdab │ "値" = cbbcca │ "軽" = cbbcbb │ "岩" = cbbcac │
"頑" = cbbbda │ "架" = cbbbcb │ "構" = cbbbbc │ "莫" = cbbbad │ "抱" =
cbbadb │ "亀" = cbbacc │ "裂" = cbbabd │ "煉" = cbadca │ "瓦" = cbadbb │
"詰" = cbadac │ "跡" = cbacda │ "筋" = cbaccb │ "枠" = cbacbc │ "割" =
cbacad │ "稲" = cbabdb │ "妻" = cbabcc │ "刻" = cbabbd │ "印" = cbaadc │
"諺" = cbaacd │ "半" = ccabda │ "円" = ccabcb │ "形" = ccabbc │ "背" =
ccabad │ "載" = ccaadb │ "！" = ccaacc │ "胆" = ccaabd │ "連" = ccbcc │ "応"
= ccbbd │ "ぜ" = cdcaba │ "定" = cdcaab │ "関" = cdcac │ "天" = cdcd │ "登"
= aaaddb │ "ギ" = aaadcc │ "ぺ" = aaadbd │ "拭" = aaacdc │ "図" = aaaccd │
"ォ" = aadcd │ "元" = abddaa │ "呼" = abdcba │ "所" = abdcab │ "今" = abdbca
│ "岸" = abdbbb │ "二" = abdbac │ "寄" = acadca │ "隊" = acadbb │ "隻" =
acadac │ "世" = acacda │ "由" = acaccb │ "砂" = acacbc │ "流" = acacad │
"細" = acabcc │ "横" = acabbd │ "伸" = accacb │ "知" = accabc │ "放" =
accaad │ "羊" = adbdb │ "権" = adbcc │ "持" = badcc │ "負" = bcbbd │ "美" =
bcdca │ "風" = bcdbb │ "波" = bcdac │ "儀" = bcdd │ "日" = bdcac │ "ゆ" =
bdcd │ "窓" = cadda │ "動" = cadcb │ "境" = cadbc │ "グ" = cadad │ "輪" =
cacdaa │ "街" = caccc │ "治" = caccba │ "都" = caccab │ "有" = cacbd │ "能"
= cbdbaa │ "信" = cbdaba │ "望" = cbdaab │ "集" = cbccaa │ "団" = cbcbba │
"便" = cbcbab │ "社" = cbcaca │ "支" = cbcabb │ "工" = cbcaac │ "委" =
cbbdaa │ "時" = cbbcba │ "汽" = cbbcab │ "午" = cbbbca │ "三" = cbbbbb │
"等" = cbbbac │ "券" = cbbada │ "買" = cbbacb │ "厳" = cbbabc │ "心" =
cbbaad │ "老" = cbadba │ "ブ" = cbadab │ "ム" = cbacca │ "ホ" = cbacbb │
"テ" = cbacac │ "床" = cbabda │ "覆" = cbabcb │ "十" = cbabbc │ "全" =
cbabad │ "頭" = cbaadb │ "面" = cbaacc │ "感" = cbaabd │ "ピ" = ccabca │
"ヴ" = ccabbb │ "進" = ccabac │ "根" = ccaada │ "神" = ccaacb │ "ポ" =
ccaabc │ "深" = ccaaad │ "套" = ccbcb │ "暗" = ccbbc │ "職" = ccbad │ "直" =
```

```
cdbd │ "多" = cdcaaa │ "以" = cdcc │ "額" = aaadda │ "参" = aaadcb │ "キ" =
aaadbc │ "尊" = aaadad │ "区" = aaacdb │ "下" = aaaccc │ "シ" = aaacbd │
"楽" = aaabcd │ "ぬ" = aadaaac │ "度" = aadaad │ "対" = aadcc │ "達" = abadd
│ "何" = abdcaa │ "役" = abdbba │ "濡" = abdbab │ "匂" = acadba │ "注" =
acadab │ "べ" = acacca │ "丈" = acacbb │ "様" = acacac │ "積" = acabcb │
"山" = acabbc │ "量" = acabad │ "無" = accaca │ "船" = accabb │ "ダ" =
accaac │ "撃" = accad │ "名" = adbda │ "河" = adbcb │ "他" = badcb │ "民" =
bbacd │ "生" = bcbbc │ "代" = bcbad │ "考" = bcdba │ "石" = bcdab │ "堤" =
bdcc │ "牧" = bdcab │ "草" = cadca │ "味" = cadbb │ "南" = cadac │ "越" =
caccaa │ "舞" = cacbc │ "町" = cacad │ "々" = cbdaaa │ "小" = cbcbaa │ "北"
= cbcaba │ "走" = cbcaab │ "復" = cbbcaa │ "保" = cbbbba │ "古" = cbbbab │
"説" = cbbaca │ "修" = cbbabb │ "商" = cbbaac │ "ず" = cbadaa │ "到" =
cbacba │ "荷" = cbacab │ "足" = cbabca │ "音" = cbabbb │ "パ" = cbabac │
"じ" = cbaada │ "歩" = cbaacb │ "ぞ" = cbaabc │ "少" = cbaaad │ "散" =
ccabba │ "巨" = ccabab │ "空" = ccaaca │ "話" = ccaabb │ "格" = ccaaac │
"助" = ccbca │ "仕" = ccbbb │ "示" = ccbac │ "づ" = ccbd │ "指" = cdbc │
"ゃ" = cdcb │ "特" = aaadca │ "廊" = aaadbb │ "晴" = aaadac │ "眠" = aaacda
│ "陸" = aaaccb │ "四" = aaacbc │ "得" = aaacad │ "身" = aaabcc │ "防" =
aaabbd │ "ぐ" = aababd │ "振" = aadaac │ "方" = aadaaab │ "広" = aadad │
"鉄" = aadcb │ "外" = abadc │ "交" = abdbaa │ "年" = acadaa │ "運" = acacba
│ "屋" = acacab │ "式" = acabd │ "敷" = acabca │ "過" = acabbb │ "ワ" =
acabac │ "着" = accaba │ "用" = accaab │ "数" = accd │ "実" = adbca │ "側" =
badd │ "ぼ" = badca │ "ひ" = bbacc │ "若" = bbabd │ "意" = bbcd │ "紹" =
bcbd │ "介" = bcbbb │ "教" = bcbac │ "ミ" = bcdaa │ "ノ" = bdcb │ "ガ" =
bdcaa │ "医" = cadba │ "リ" = cadab │ "差" = cacbb │ "安" = cacac │ "素" =
cbcaaa │ "塔" = cbbbaa │ "線" = cbbaba │ "内" = cbbaab │ "送" = cbacaa │
"後" = cbabba │ "港" = cbabab │ "ざ" = cbaaca │ "市" = cbaabb │ "取" =
cbaaac │ "ッ" = ccad │ "利" = ccabaa │ "明" = ccaaba │ "行" = ccaaab │ "ご"
= ccbba │ "目" = ccbab │ "駅" = cdbb │ "降" = cdd │ "前" = aaadba │ "ぎ" =
aaadab │ "友" = aaacca │ "奏" = aaacbb │ "重" = aaacac │ "不" = aaabcb │
"場" = aaabbc │ "川" = aaabad │ "決" = aababc │ "自" = aabaad │ "馬" =
aadaab │ "員" = aadaaaa │ "ァ" = aadac │ "ド" = aadca │ "ぶ" = aadd │ "家" =
ababd │ "合" = abadb │ "—" = acacaa │ "分" = acabba │ "気" = acabab │ "祭" =
acaad │ "部" = accc │ "マ" = accaaa │ "ほ" = baabd │ "オ" = bbacb │ "ろ" =
bbabc │ "び" = bbaad │ "立" = bbcc │ "水" = bcbc │ "築" = bcbba │ "乗" =
bcbab │ "雨" = bdd │ "通" = cadaa │ "ょ" = cacba │ "入" = cacab │ "ね" =
cbbaaa │ "主" = cbabaa │ "任" = cbaaba │ "司" = cbaaab │ "地" = ccac │ "一"
= ccaaaa │ "口" = ccbaa │ "会" = ccd │ "べ" = cdba │ "ク" = aaadaa │ "げ" =
aaacba │ "海" = aaacab │ "中" = aaabd │ "む" = aaabca │ "道" = aaabbb │ "ョ"
= aaabac │ "事" = aababb │ "物" = aabaac │ "み" = aabad │ "チ" = aadab │
"言" = ababc │ "タ" = abaad │ "手" = abada │ "ば" = acabaa │ "サ" = acaac │
"車" = accb │ "堂" = adbb │ "建" = add │ "フ" = baabc │ "出" = baaad │ "せ"
= badb │ "ア" = bbaca │ "思" = bbabb │ "め" = bbaac │ "上" = bbad │ "エ" =
bbcb │ "者" = bcbaa │ "大" = caad │ "聖" = cacaa │ "彼" = cbaaaa │ "ど" =
```

```
ccc │ "や" = aaaad │ "ジ" = aaacaa │ "ウ" = aaabba │ "レ" = aaabab │ "カ" =
aababa │ "人" = aabaab │ "イ" = aabac │ "見" = aabd │ "ト" = aadb │ "ラ" =
ababb │ "つ" = abaac │ "ス" = abda │ "え" = acaab │ "ち" = acd │ "ん" = adba
│ "ル" = adc │ "け" = baabb │ "「" = baaac │ "」" = baad │ "お" = bada │
"・" = bbaba │ " " = bbaab │ "  " = bbca │ "わ" = bbd │ "も" = bcc │ "あ" =
bdb │ "さ" = caac │ "よ" = cda │ "だ" = aaaac │ "く" = aaabaa │ "ン" = aabc
│ "そ" = aabaaa │ "ま" = ababa │ "き" = abaab │ "ー" = abac │ "り" = acaaa │
"す" = baaba │ "れ" = baaab │ "ら" = baac │ "う" = bbaaa │ "っ" = bda │ "か"
= caab │ "で" = aaaab │ "こ" = aabb │ "る" = abaaa │ "が" = abc │ "。" = acb
│ "な" = ada │ "、" = baaaa │ "は" = bac │ "て" = bbb │ "を" = bca │ "と" =
caaa │ "し" = cab │ "い" = aaaaa │ "に" = aac │ "た" = abb │ "の" = bab │

Textlänge: 4577 · Encodete Länge: 20363 Perlen · Gesamtkosten (∑): 37664mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck-custom-0.txt ---

│ "S" = cce │ "U" = ch │ "Q" = ccd │ "N" = cg │ "f" = ccab │ "?" = ccc │
"h" = cf │ "x" = ag │ "." = ccaa │ "g" = ccb │ "b" = ce │ "," = af │ "v" =
cac │ "c" = cd │ "d" = aad │ "p" = ae │ "q" = cab │ "l" = f │ "n" = aac │
"r" = ad │ "s" = caa │ "m" = cb │ "o" = e │ "t" = aab │ "u" = ac │ "a" = d
│ "i" = aaa │ "e" = ab │ " " = b │

Textlänge: 865 · Encodete Länge: 1746 Perlen · Gesamtkosten (∑): 3722mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck-custom-1.txt ---

│ "q" = aabbaab │ "A" = ababaab │ "j" = abbaaab │ "S" = aabbaaa │ "L" =
aabbab │ "y" = ababaaa │ "k" = ababab │ "v" = abbaaaa │ "b" = abbaab │ "."
= aaaaaab │ "," = aabaab │ "g" = aabbb │ "p" = ababb │ "c" = abbab │ "l" =
aaaaaaa │ "n" = aaaaab │ "d" = aabaaa │ "i" = aabab │ "u" = abaab │ "r" =
abbb │ "m" = aaaab │ "s" = abaaa │ "o" = baab │ "a" = aaab │ "t" = baaa │
"e" = bab │ " " = bb │

Textlänge: 591 · Encodete Länge: 2530 Perlen · Gesamtkosten (∑): 3544mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!

--- AUSGABE BEI DATEI ./A1_Schmucknachrichten/schmuck-custom-2.txt ---

│ "q" = aabab │ "A" = aabaa │ "j" = aabbabab │ "S" = aabbabaa │ "L" =
aabbabb │ "y" = aabbaa │ "k" = aabbb │ "v" = aaaab │ "b" = aaaaaab │ "." =
aaaaaaa │ "," = aaaaabab │ "g" = aaaaabaa │ "p" = aaaaabb │ "c" = aaab │
"l" = ab │ "n" = babab │ "d" = babaa │ "i" = babbab │ "u" = babbaa │ "r" =
babbbab │ "m" = babbbaab │ "s" = babbbaaa │ "o" = babbbb │ "a" = baaab │
"t" = baaaa │ "e" = baab │ " " = bb │

Textlänge: 591 · Encodete Länge: 3022 Perlen · Gesamtkosten (∑): 0mm

Überprüfung nach Präfixdopplungen beendet - Code ist präfixfrei!
```