Aufgabe 1 b: Schmucknachrichten greedy Lösung

Teilnahme-ID: 74993

Bearbeiter dieser Aufgabe:
Mathieu de Borman

28. April 2025

Inhaltsverzeichnis

1 Lösungsidee **1**
1.1 Gleiche Häufigkeiten 2
1.2 Ungleiche Häufigkeiten 3
1.3 Durchschnittliche Zeichentiefe 3
1.4 Baumkorrekturen 4
1.5 Optimierungen 5

2 Umsetzung **5**

3 Komplexitätsanalyse **5**

4 Laufzeitanalyse **6**

5 Beispiele **6**

6 Quelltext **9**

1 Lösungsidee

In dieser Überlegung wird die Codierung wieder als gewurzelter Baum betrachtet. Jede Kante steht für eine Perle und jedes Blatt für ein Zeichen im Ursprungstext. Der Pfad von der Wurzel zu dem Zeichen stellt das Codewort des jeweiligen Zeichens dar. Das Gewicht jeder Kante entspricht der Summe aller Perlendurchmesser auf dem Pfad bis zur Wurzel. Das Gewicht eines inneren Knotens entspricht der Summe der Gewichte seiner Kinder. Das Gewicht eines äußeren Knotens entspricht der Häufigkeit des Zeichens, welcher diesem Knoten zugewiesen wurde.

Die Idee dieser Lösung ist es, für jede Zeichenhäufigkeit in etwa zu ermitteln, wie Tief diese im Baum sein muss, um diese Anschließend einem Blatt dieser Tiefe zuzuweisen. Durch die Struktur des Baums ist dann Garantiert, dass die Codierung Präfixfrei ist. Mit Tiefe meine ich hier nicht die Anzahl an Kanten von Wurzel zu dem jeweiligen Knoten, sondern das Gewicht der Kante zu dem jeweiligen Knoten, d.h. die Summe der Perlendurchmesser mit dem der Knoten codiert ist. Dadurch kann es z.B. sein, dass ein Knoten zwar weniger Kanten Abstand zur Wurzel hat, die Perlen, die diese Kanten darstellen allerdings so groß sind, dass es Tiefer im Baum liegt. Hier eine Visualisierung für Perlen mit den Durchmessern 1 und 3.

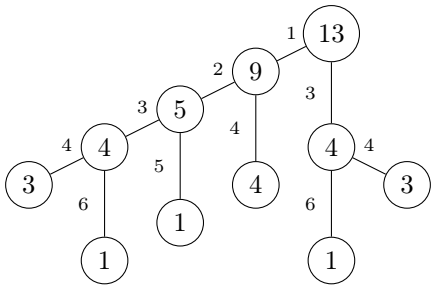

Abbildung 1: Ein Baum mit Perlendurchmessern 1 und 3

1.1 Gleiche Häufigkeiten

In dem Fall, dass alle Zeichen die gleiche Häufigkeit haben, kann man in Polynomialzeit einen optimalen Baum für diese Konstruieren. Zuerst wird ermittelt wie viele innere Knoten benötigt werden. Wenn p die Anzahl an Perlen und $|s|$ die Anzahl an verschiedenen Zeichen sind, dann kann man dafür folgende Formel benutzen.

$$\left\lceil \frac{|s|-1}{p-1} \right\rceil$$

Allerdings sind Bäume mit der minimalen Anzahl an inneren Knoten nicht optimal. Z.B. wenn es drei verschiedene Zeichen gibt und die Perlen die Durchmesser 1, 1 und 10 haben. Deswegen braucht es manchmal noch weitere innere Knoten. Sei t die Tiefen und t_i die Tiefe des i-ten Knoten. Sei d die Durchmesser der Perlen in aufsteigender Reihenfolge und d_i der Durchmesser der i-ten Perle. d_p ist also die größte Perle. Indem man bei einem Blattknoten i mindestens ein Kind Anhängt, wird das Codewort des Zeichen des nun ehemaligen Blattknotens um d_1 länger. Allerdings können alle Zeichen mit einem Blattknoten mit einer größeren Tiefe, den neu entstandenen Blattknoten zugewiesen werden. Die durch das erweitern entstandene Differenz in der Länge des codierten Texts beträgt also d_1 minus die Differenz der Blätter, die neu zugeordnet werden. Und das Lässt sich in Polynomialzeit berechnen. Es werden immer die Blätter mit der kleinsten Tiefe erweitert. Das liegt daran, dass ein innerer Knoten immer mindestens Zwei Kinder hat. Sei k die Differenz der Tiefe zwischen dem höchsten Blattknoten und einem anderen Knoten. Wenn jedes der beiden Blattknoten Kinder angehängt bekommt, dann sind alle Zeichen, die Nachfahren vom höheren Knoten sind um k höher im Baum, als die Nachfahren vom anderen Knoten. Wenn man also den höheren Knoten erweitert, dann gibt es mehr Zeichen, die um k kürzer codiert sind. Daher ist es, wenn Erweitern die Länge des codierten Texts verkürzt, immer optimal dem höchsten Blattknoten Kinder zu geben (Abschnitt 1.1 der iterativen Lösung belegt das nochmals). Hier ein Beispiel mit 10 verschiedenen Zeichen der Häufigkeit 1 und den Perlendurchmessern 1, 1 und 3.

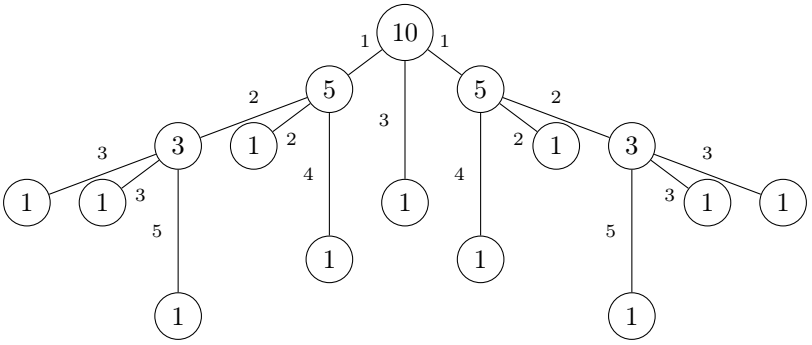

Abbildung 2: Baum mit einer codierten Textlänge von 37 mm

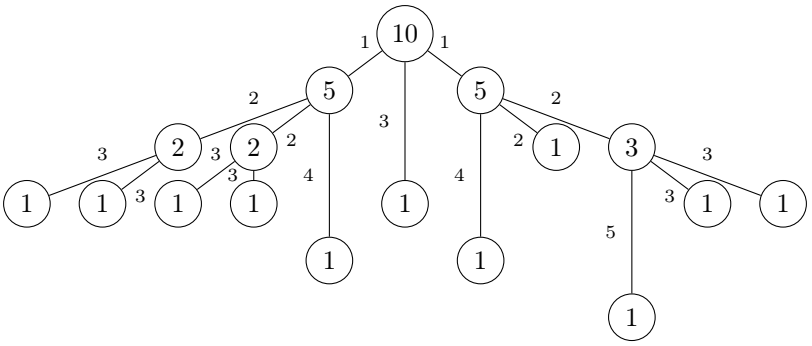

Abbildung 3: Baum mit einer codierten Textlänge von 36 mm

1.2 Ungleiche Häufigkeiten

Wenn man bei einem einem optimalen Baum, alle Zeichen im Teilbaum eines inneren Knotens als gleiche Zeichen denkt, dann ist der daraus resultierende Baum, für den entstandenen Text auch optimal. Zumindest vermute ich das, habe aber leider keinen Beweis. Für gleiche Perlengrößen ist das jedenfalls korrekt, da die Huffman-Codierung diese Idee als Zentralen Teil des Algorithmus nutzt. Hier ein Beispiel wie ein solches Zusammenfassen aussehen könnte. Die Perlendurchmesser sind 1 und 3.

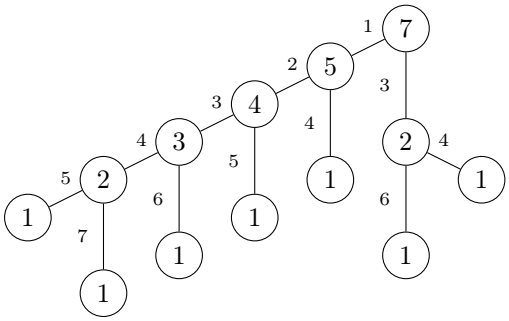

Abbildung 4: Beispieltext: "abcdefg"

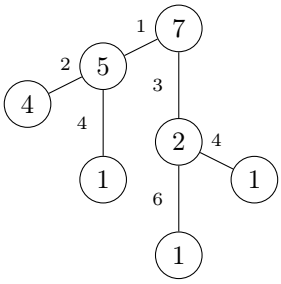

Abbildung 5: Beispieltext: "aaaaefg"

Ein Algorithmus für ungleiche Zeichenhäufigkeiten, könnte also sein, dass man einen Baum konstruiert, der für die gleiche Textlänge aber nur einmal vorkommenden Zeichen optimal ist. Anschließend ersetzt man innere Knoten durch Zeichen, deren Häufigkeit gleich der gewichte der inneren Knoten ist. Der Algorithmus würde z.B., wie in Abbildung 4&5 gezeigt, einen optimalen Baum für den Text "aaaaefg"erzeugen. Allerdings gibt es in manchen Bäumen nicht für jede mögliche Zeichenhäufigkeit auch einen entsprechend groß gewichteten inneren Knoten. Die optimale Version des Baums aus den Abbildungen 2&3 würde z.B. keinen Knoten mit dem Gewicht 3 haben. Daher funktioniert dieser Algorithmus nur für sehr spezielle Fälle.

1.3 Durchschnittliche Zeichentiefe

Statt die genauen Gewichte der Knoten zu betrachten, legt mein Algorithmus eher den Fokus darauf, wie groß der durchschnittliche Abstand zwischen einem inneren Knoten und den Blättern im Teilbaum des Knotens ist. Beim Zusammenfassen von Zeichen ändert sich ihre durchschnittliche Tiefe. Addiert man zur Tiefe eines Knotens die durchschnittliche Tiefe eines Baums mit gleich vielen, aber unterschiedlichen Zeichen, und man dieses Zeichen nun mehrmals zählt, bleibt der Wert konstant. Hier ein Beispiel an Abbildung 4&5. Die Durchschnittliche Tiefe eines Blattknotens in Abbildung 4 ist $\frac{37}{7}$. Die durchschnittliche Tiefe bei Abbildung 5 ist 4. Wenn man nun die durchschnittliche Tiefe für einen Baum mit 4 Zeichen, also $\frac{15}{4}$ zusätzlich betrachten, dann hat der Baum eine neue Durchschnittliche Tiefe von $(4(2+\frac{15}{4})+14):7=(8+14+15):7=\frac{37}{7}$. Durch Minimierung der Differenz zwischen durchschnittlicher Tiefe im Baum mit gleichen Wahrscheinlichkeiten und der nachberechneten Tiefe im tatsächlich verwendeten Baum, könnte so eine optimale Länge für den codierten Text gefunden werden. Diese Methode deckt alle möglichen Eingabetexte ab.

Ich habe aktuell leider keine Lösung, mit der diese Idee in einem effizienten Algorithmus mit optimalen Ergebnissen umgesetzt werden kann. Ich habe dafür aber eine Heuristik.
Ich nenne den Wert, der durchschnittlichen Tiefe für einen Baum mit gleichen Wahrscheinlichkeiten und gleicher Zeichenanzahl, wie das Zeichen selbst, Erwartungswert. Der Erwartungswert z.B. für 4 in Abbildung 5 ist $\frac{15}{4}$. Statt sich dem Durchschnitt Global anzunähern, wird für jedes Zeichen bestimmt, wie dessen Tiefe sein muss, damit die Summe aus dessen Tiefe und dessen Erwartungswert möglichst nah an dem Erwartungswert des Gesamten Texts ist. Dieser Tiefe, also der Differenz zwischen Texterwartungswert und Zeichenerwartungswert werden dann alle Zeichen dieser Häufigkeit zugewiesen. Deswegen ist das die greedy Lösung.
Hier mal eine Visualisierung. Die Höhe eines Strichs entspricht dessen Wert. Der gestrichelte Strich stellt den Erwartungswert für die Textlänge bzw. dem Wurzelknoten dar. Die durchgezogenen Striche stellen die Summe aus Erwartungswert und Tiefe der jeweiligen Zeichen dar.

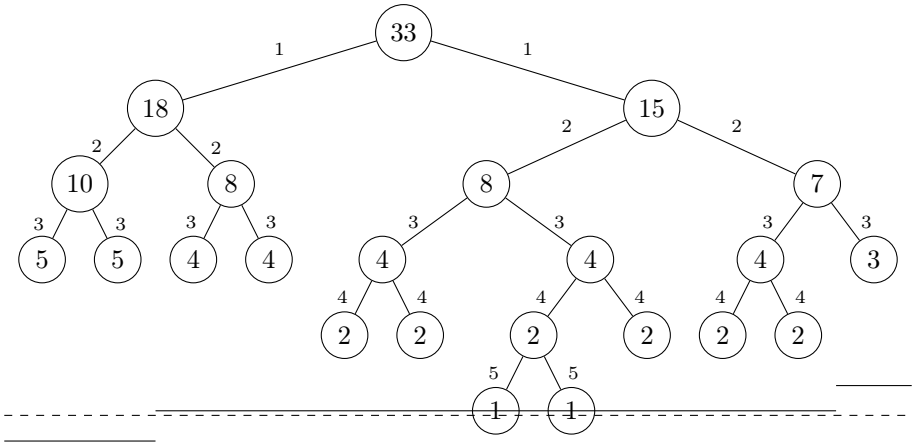

Abbildung 6: Ein optimaler Baum für schmuck0.txt

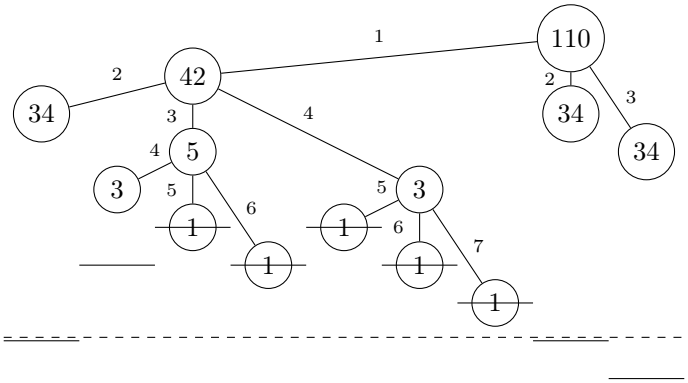

Abbildung 7: Ein optimaler Baum für schmuck3.txt

1.4 Baumkorrekturen

Die Differenz aus Texterwartungswert und Zeichenerwartungswert können nur als Richtwert verwendet werden, da diese nicht immer einen so möglichen Baum berechnen. Das ist der Fall wenn mehr Zeichen einer Tiefe zugewiesen werden, als es Knoten in dieser Tiefe geben kann. Das kann man z.B. sehr gut an den 34 Häufigen Zeichen aus Abbildung 7 gut erkennen. Anhand der der Differenzen der Erwartungswerte würden eigentlich alle Zeichen mit der Häufigkeit 34 der Tiefe 2 zugeordnet werden. Allerdings gibt es maximal nur zwei Knoten dieser Tiefe. Der Algorithmus reagiert darauf, indem es eigentlich nicht zuweisbare Zeichen der nächsten Tiefe zuweisen. Dadurch entsteht allerdings ein anderes Problem. Manchmal entsteht ein abgeschlossener Baum, ohne dass alle Zeichen einem Blattknoten zugewiesen sind. Abgeschlossen heißt, dass jedem Blattknoten ein Zeichen zugewiesen wurde.

Um nun die restlichen Zeichen zuweisen zu können, müssen Blattknoten zu inneren Knoten gemacht werden. Der Algorithmus erweitert in dem Fall immer die größten Blattknoten.

1.5 Optimierungen

Manchmal gibt es mehr Blattknoten als verschiedene Zeichen. Das ist z.B. der Fall, wenn die tiefsten Zeichen, aufgrund der Erwartungswerte tiefer liegen, als diese es sein müssten. In dem Fall, dass mehr Blattknoten als Zeichen vorkommen, wird die Anzahl an inneren Knoten minimiert, indem die tiefsten inneren Knoten zu Blattknoten gemacht werden. Die minimale Anzahl an inneren Knoten ist allerdings nicht immer optimal, wie man in Abschnitt 1.1 erkennen kann. Diese Optimierung sorgt allerdings in der Regel für leicht bessere codierte Textlängen.

Manchmal sind Perlen so groß, dass diese garantiert nicht verwendet werden. Um also die Laufzeit zu verbessern, wird die Heuristik aus der ILP Lösung verwendet, um die maximale Codewortlänge zu ermitteln. Alle Perlen, die größer als diese Codewortlänge sind, werden entfernt.
Es werden gleiche Zeichenhäufigkeiten zusammengefasst und gemeinsam betrachtet. Dadurch müssen z.B. weniger Erwartungswerte berechnet werden.

2 Umsetzung

Die Lösung wurde in JavaScript Umgesetzt. Zuerst wird ermittelt, welche Perlen nicht betrachtet werden müssen. Anschließend werden die Zeichenhäufigkeiten in der Funktion *getChars* mithilfe einer Hashmap ermittelt. In derselben Funktion werden auch die Zeichenhäufigkeiten zusammengefasst und die Erwartungswerte für jede Häufigkeit gespeichert. Die Ermittlung der Erwartungswerte bzw. alle Berechnungen werden nicht anhand von Baum ähnlichen Datenstrukturen, sondern an ein zwei Listen getätigt, die in etwa das Gleiche wie a und b aus der ILP Lösung speichern. Die Liste *outer* speichert, wie viele Blattknoten der jeweiligen Tiefe existieren. Die Liste *inner* speichert wie viele innere Knoten der jeweiligen Tiefe existieren. Diese Methode ermöglicht es den Baum mithilfe von wenigen Additionen und Subtraktionen zu vergrößern oder zu verkleinern. Hier ein Beispiel für das Erweitern eines Blattknoten mit der Tiefe 4 und den Perlen 1, 1, 2 und 4.

Tiefe	0	1	2	3	4	5	6	7	8	9
outer	0	0	2	3	5	10	10	5	4	2
inner	1	2	3	3	4	2	0	0	0	0

Abbildung 8: vor dem Erweitern

Tiefe	0	1	2	3	4	5	6	7	8	9
outer	0	0	2	3	4	12	11	5	5	2
inner	1	2	3	3	5	2	0	0	0	0

Abbildung 9: nach dem Erweitern

Das Zuweisen der Zeichen wird in der Funktion *greedyTree* bewerkstelligt. Das Erstellen der Codetabelle in der Funktion *decompress* wird durch den gleichen Algorithmus umgesetzt wie in der ILP Lösung. Das Ermitteln der Länge des codierten Texts ist in der Funktion *encodedLength* umgesetzt.

3 Komplexitätsanalyse

Ich verwende das Wort Komplexität stellvertretend für Laufzeit- und Speicherkomplexität, um die Lesbarkeit zu erhöhen. Wenn sich die Laufzeitkomplexität von der Speicherkomplexität unterscheidet, dann spezifiziere ich das. Die Komplexität wird in Abhängigkeit von p, d, der oberen Grenze für die Tiefe k und der Länge des Eingabetexts l formuliert. k wird hier auch als Variable für die Länge von *outer* bzw. *inner* verwendet, da die Länge der beiden Listen nicht größer als k wird. Das Einlesen der Durchmesser und die Eliminierung der größten Durchmesser haben eine Komplexität von $\mathcal{O}(p \log p + kp + d_p)$.

Das Zählen und Zusammenfassen der Zeichenhäufigkeiten hat eine Komplexität von $\mathcal{O}(l \log l)$. Das Berechnen der Erwartungswerte hat eine Komplexität von $\mathcal{O}(lk + lp)$). Das Zuweisen der Zeichen hat eine Komplexität von $\mathcal{O}(l + k)$. Das Erstellen der Codetabelle hat eine Komplexität von $\mathcal{O}(lk + l \log l)$. Das berechnen der codierten Textlänge hat eine Komplexität von $\mathcal{O}(l+k)$. Insgesamt hat der Algorithmus eine Komplexität von $\mathcal{O}(p \log p + l \log l + lk + pk + d_p)$. Durch gleiche Betrachtungen von k, wie sie schon in der ILP Lösung getätigt wurden, ist die Komplexität für das Programm $\mathcal{O}(p \log p + l^2 d_p + l d_p p)$. Allerdings ist die obere Grenze Präziser mit $k = d_2 \log l$, also einer Komplexität von $\mathcal{O}(p \log p + l d_2 \log l + p d_2 \log l + d_p)$ formuliert.

4 Laufzeitanalyse

Da die Beispieleingaben in sehr kurzer Zeit gelöst werden, habe ich auch Eingaben mit bis zu 150.000 verschiedenen Zeichen getestet. Durch die Analyse, für welche Prozesse wie viel Zeit benötigt wurde, lassen sich in diesem Fall überraschende Ergebnisse erzielen. Z.B. nimmt das Zählen der Zeichenhäufigkeiten etwa die Hälfte der Laufzeit ein. Am Überraschendsten ist allerdings, dass die mit Abstand größte Zeit ver(sch)wendet wird, um die Codetabelle anzuzeigen! Hier ein z.B. ein Ausschnitt aus dem Firefox Profiler (das Laufzeitanalyse Tool von Firefox).

Der erste gelbe Abschnitt (ca. 400ms), zeigt die Prozesse vom Algorithmus, d.h. Lösung berechnen, Codetabelle erstellen und so weiter. Der lila Abschnitt (ca. 2,5s) zeigt die Prozesse, die benötigt werden, um das Ergebnis anzuzeigen. Wir man erkennen kann, wird die Lösung in nur einem Bruchteil der Zeit ermittelt, die zwischen Start und angezeigter Lösung vergeht. Bei Google Chrome ist der Browser abgestürzt und bei Microsoft Edge habe ich das Programm nach einer Minute abgebrochen. Sobald ich die Codetabelle zwar berechnet, aber nicht ausgegeben habe, wurden die Ergebnisse innerhalb der selben Sekunde angezeigt. Die Laufzeit, die mein Programm angibt, ist deswegen nur für das Berechnen der Ergebnisse. Mein Programm misst z.B. bei der Eingabedatei mit 150.000 Zeichen nur die ersten 400ms. Die mit Abstand beste Optimierung für meine Lösung wäre dementsprechend die Programmiersprache zu wechseln.

5 Beispiele

Wir rufen nun das JavaScript-Programm mit den verschiedenen BwInf-Eingabedateien auf. Das eigene Beispiel "schmuck13.txt" soll zeigen, dass das Programm auch mit in etwa allen Unicode Zeichen auf einmal umgehen kann. Die Beispieldatei enthält unter anderem auch Steuerzeichen um auf 150.000 verschiedene Zeichen zu kommen. Diese Datei ist in demselben Ordner wie die Programmdatei.
Das Programm wird mit Hilfe des Browsers ausgeführt. Mithilfe des "Datei Hochladen" Knopfs kann man Beispieleingaben auswählen. Die Ausgabe erscheint (irgendwann) auf der Webseite. Bei der Codetabelle steht immer der Index der jeweiligen Perle statt der Länge der Perle. Unter "Solution Data" stehen die komprimierten Lösungsdaten, d.h. die Werte von *outer*. Das Programm terminiert für alle getesteten Eingaben in weniger als 500ms auf einem gewöhnlichen PC. Für "schmuck1.txt" und "schmuck5.txt" bis "schmuck13.txt" habe ich die Codetabelle weggelassen, um diesen Abschnitt kurz zu halten. Die Ergebnisse von " schmuck1.txt" bis "schmuck13.txt" sind auch im Ordner der Programmdatei zu finden. Um diesen Algorithmus mit den anderen Lösungen für die Aufgabe 1 vergleichen zu können, habe ich hier alle Ergebnisse tabellarisch abgebildet.

Eingabedatei	Lösung Teilaufgabe a		ILP Lösung		greedy Lösung	
	Nachrichtenlänge	Laufzeit	Nachrichtenlänge	Laufzeit	Nachrichtenlänge	Laufzeit
schmuck0.txt	113	0s	113	0.045s	113	0.001s
schmuck00.txt	372	0.001s	372	0.063s	373	0.002s
schmuck01.txt	1150	0s	1150	0.043s	1223	0s
schmuck03.txt	406	0.001s	406	0.038s	406	0s
schmuck1.txt	197	0.001s	191	0.047s	191	0.002s
schmuck2.txt	145	0.001s	135	0.041s	135	0s
schmuck3.txt	279	0s	279	0.037s	279	0.002s
schmuck4.txt	154	0.001s	137	0.034s	137	0s
schmuck5.txt	4010	0.001s	3162	0.051s	3185	0.002s
schmuck6.txt	244	0s	234	0.075s	234	0.001s
schmuck7.txt	153144	0.006s	134559	0.151s	134563	0.038s
schmuck8.txt	3615	0s	3287	0.039s	3302	0.001s
schmuck9.txt	41056	0.004s	36597	0.089s	36616	0.002s
schmuck10.txt	44262	0.002s	37230	0.088s	37402	0.031s
schmuck11.txt	424824913	0.003s	84886	0.131s	89891	0.107s
schmuck13.txt	7458860	0.258s	7458860	0.442s	7691560	0.274s

schmuck1.txt

```
Runtime: 0.002s
Pearllengths in mm: 1,1,2
Messagelength in mm: 191
Solution Data: 0,0,1,6,11,6,1
```

schmuck2.txt

```
Runtime: 0s
Pearllengths in mm: 1,5
Messagelength in mm: 135
Solution Data: 0,1,0,0,0,0,0,0,0,0,0,3,1,1,1,2
a: [0]
 : [1,1,0]
b: [1,0,0,0,0,0,0]
c: [1,0,1]
d: [1,0,0,1]
e: [1,0,0,0,1]
f: [1,0,0,0,0,1]
g: [1,1,1]
h: [1,0,0,0,0,0,1]
```

schmuck3.txt

```
Runtime: 0.002s
Pearllengths in mm: 1,2,3
Messagelength in mm: 279
Solution Data: 0,0,2,1,1,2,2,1
a: [0,0]
b: [1]
c: [0,1]
 : [2,0]
d: [0,2,0]
e: [2,1]
f: [0,2,1]
g: [2,2]
h: [0,2,2]
```

schmuck4.txt

```
Runtime: 0s
Pearllengths in mm: 1,5
Messagelength in mm: 137
Solution Data: 0,0,0,0,0,0,0,1,4,1,2,3,3
a: [0,1,0]
b: [0,0,1,0]
c: [1,0,0,0]
d: [0,0,0,0,0,0,0,0]
e: [0,0,0,1]
f: [0,0,0,0,1]
g: [1,1]
h: [0,0,0,0,0,1]
i: [0,1,1]
j: [0,0,0,0,0,0,1]
k: [1,0,1]
l: [0,0,1,1]
m: [1,0,0,1]
n: [0,0,0,0,0,0,0,1]
```

schmuck5.txt

```
Runtime: 0.002s
Pearllengths in mm: 1,1,2,3,4,5,6
Messagelength in mm: 3185
Solution Data: 0,0,1,11,7,7,7,6,2
```

schmuck6.txt

```
Runtime: 0.001s
Pearllengths in mm: 1,2,3
Messagelength in mm: 234
Solution Data: 0,0,0,0,1,9,13,9,2
```

schmuck7.txt

```
Runtime: 0.038s
Pearllengths in mm: 1,1,1,1,1,1,1,2,3,4
Messagelength in mm: 134563
Solution Data: 0,4,19,23,21,8,5,2
```

schmuck8.txt

```
Runtime: 0.001s
Pearllengths in mm: 1,1,2,2,3
Messagelength in mm: 3302
Solution Data: 0,0,0,2,11,79,127,79,23
```

schmuck9.txt

```
Runtime: 0.002s
Pearllengths in mm: 1,2,3,4
Messagelength in mm: 36616
Solution Data: 0,0,0,0,0,5,13,10,21,27,74,96,161,134,86,47
```

schmuck10.txt

```
Runtime: 0.031s
Pearllengths in mm: 1,2,2,3,3,3,4,4,4,4,5,5,5,5,5,6,6,6,6,6,6,7,7,7,7,7,7,7
Messagelength in mm: 37402
Solution Data: 0,1,0,0,1,26,60,87,106,132,142,126,37
```

schmuck11.txt

```
Runtime: 0.107s
Pearllengths in mm: 1,5,23,27
Messagelength in mm: 89891
Solution Data: 0,0,0,0,0,0,0,0,0,0,0,0,3,7,4,7,3,4,61,24,24,36,50,5,1,1,2,3,4,5,6,8,
11,15,20,23,24,31,39,53,65,24,24,36,49,2
```

schmuck13.txt

```
Runtime: 0.274s
Pearllengths in mm: 1,1
Messagelength in mm: 7691560
Solution Data: 0,0,0,0,0,0,0,0,0,0,0,949,75,0,0,0,579,38925,55086,0,0,0,0,0,24325,30710
```

6 Quelltext

```
//funktion um die Durchmesserdaten zu konvertieren
function compactPearls(diameters) {
 //sortiere die Durchmesser
 diameters.sort((a, b) => a - b)
 let pearlAmount = []
 let pearlSize = []
 let lastSize = 0
 //gehe durch jeden Durchmesser durch und speichere dir
 //wie viele von der jeweiligen größe existieren
 for (let i = 0; i < diameters.length; i++) {
  if (lastSize == diameters[i]) {
   pearlAmount[pearlAmount.length - 1]++
  } else {
   lastSize = diameters[i]
   pearlSize.push(lastSize)
   pearlAmount.push(1)
  }
 }
 return [pearlAmount, pearlSize]
}
//funktion um k erstmals zu bestimmen
function heuristic([pearlAmount, pearlSize], message) {
 //erstell die Liste mit den Codewortlängen
 let len = pearlSize[pearlSize.length - 1] + 1
 let wordLens = Array(len).fill(0)
 wordLens[0] = 1
 //gehe durch alle Codewortlängen durch, bis es genügend verschiedene Codewörter einer Länge gibt
 let k = 0
 while (wordLens[k % len] < message.length) {
  for (let i = 0; i < pearlAmount.length; i++) {
   wordLens[(k + pearlSize[i]) % len] += wordLens[k % len] * pearlAmount[i]
  }
  wordLens[k % len] = 0
  k++
 }
 return k
}
//funktion um den Baum zu bearbeiten
function grow(outer, changeInd, [pearlAmount, pearlSize], inner, amount = 1) {
 inner[changeInd] += amount
 outer[changeInd] -= amount
 for (let i = 0; i < pearlSize.length; i++) {
  while (outer.length <= changeInd + pearlSize[i]) {
   outer.push(0)
```

```
   inner.push(0)
  }
  outer[changeInd + pearlSize[i]] += amount * pearlAmount[i]
 }
 while (outer[outer.length - 1] == 0) {
  outer.pop()
  inner.pop()
 }
}
//funktion um schnell einen Baum für gleiche Zeichenhäufigkeiten zu konstruieren
function fastEqualTree(pearls, treeSize, diameters) {
 let outer = [1]
 let inner = [0]
 //berechne die Anzahl an inneren Knoten
 let innerSum = Math.ceil((treeSize - 1) / (diameters.length - 1))
 let ind = 0
 //füge so viele Knoten in den Baum ein
 for (let i = 0; i < innerSum; i++) {
  while (outer[ind] == 0) {
   ind++
  }
  grow(outer, ind, pearls, inner)
 }
 return [outer, inner]
}
function optimiseEqualTree([outer, inner], [pearlAmount, pearlSize], treeSize) {
 //Zähle die Blätter des Baums
 let outerSum = 0
 for (let i = 0; i < outer.length; i++) {
  outerSum += outer[i]
 }
 //berechne wie viel besser ein Baum mit mehr inneren Knoten wäre
 let destructed = [...outer]
 while (outerSum > treeSize) {
  destructed[destructed.length - 1]--
  outerSum--
  while (destructed[destructed.length - 1] == 0) {
   destructed.pop()
  }
 }
 let minInd
 for (let i = 0; i < outer.length; i++) {
  if (outer[i] > 0) {
   minInd = i
   break
  }
 }
 let dif = minInd
 for (let i = 0; i < pearlSize.length; i++) {
  dif -= Math.max(0, destructed.length - 1 - minInd - pearlSize[i])
  if (destructed.length - 1 - minInd - pearlSize[i] > 0) {
   destructed[destructed.length - 1] -= pearlAmount[i]
  }
  while (destructed[destructed.length - 1] == 0) {
   destructed.pop()
  }
 }
 //falls es besser wäre, füge einen inneren Knoten ein
```

```
 if (dif < 0) {
  grow(outer, minInd, [pearlAmount, pearlSize], inner)
 }
 //gebe zurück, ob optimiert wurde
 return dif < 0
}
//funktion um die erwartbare Tiefe eines Baums zu berechnen
function getAverageDepth(pearls, treeSize, diameters) {
 //erstelle einen otimalen Baum für gleiche Wahhrscheinlichkeiten
 let tree = fastEqualTree(pearls, treeSize, diameters)
 while (optimiseEqualTree(tree, pearls, treeSize)) { }
 //berechne die durchschnittliche Tiefe
 let [compact, hidden] = tree
 let ind = 0
 let sum = 0
 for (let i = 0; i < treeSize; i++) {
  while (compact[ind] == 0) {
   ind++
  }
  sum += ind
  compact[ind]--
 }
 return (sum / treeSize)
}
//funktion um die länge des codierten Texts zu ermitteln
function encodedLength(outer, [charAmount, charSize, charDepth, charDetail]) {
 outer = [...outer]
 charAmount = [...charAmount]
 let score = 0
 let charInd = 0
 let treeInd = 0
 while (charAmount[charAmount.length - 1] > 0) {
  while (outer[treeInd] == 0) {
   treeInd++
  }
  while (charAmount[charInd] == 0) {
   charInd++
  }
  let change = Math.min(outer[treeInd], charAmount[charInd])
  score += treeInd * charSize[charInd] * change
  outer[treeInd] -= change
  charAmount[charInd] -= change
 }
 return score
}
//funktion um zu ermiteln wie häufig es welches zeichen es gibt
//und wie viele Zeichen mit der Häufigkeit existieren
function getChars(pearls, message, d) {
 //ermittle wie häufig jedes Zeichen vorkommt
 let chars = {}
 for (let i = 0; i < message.length; i++) {
  chars[message.codePointAt(i)] = chars[message.codePointAt(i)] + 1 || 1
 }
 chars = Object.entries(chars)
 for (let i = 0; i < chars.length; i++) {
  chars[i][0] = String.fromCodePoint(chars[i][0])
 }
 //ermittle welche Zeichen die gleiche Häufigkeit haben
```

```
 chars.sort((a, b) => b[1] - a[1])
 let charAmount = []
 let charFrequency = []
 let lastFrequency = 0
 for (let i = 0; i < chars.length; i++) {
  if (lastFrequency == chars[i][1]) {
   charAmount[charAmount.length - 1]++
  } else {
   lastFrequency = chars[i][1]
   charFrequency.push(lastFrequency)
   charAmount.push(1)
  }
 }
 //füge die zu erwartende Distanz zur Tiefe des Baums hinzu
 let charDepth = []
 for (let i = 0; i < charFrequency.length; i++) {
  charDepth.push(getAverageDepth(pearls, charFrequency[i], d))
 }
 return [charAmount, charFrequency, charDepth, chars]
}
//funktion um einen Baum für ungleiche Häufigkeiten zu konstruieren
function greedyTree(pearls,[charAmount,charFrequency,charDepth,charDetail],message,diameters) {
 let upperLimit = getAverageDepth(pearls, message.length, diameters)
 let outer = [1]
 let assigned = [0]
 let inner = [0]
 let ind = 0
 //weise jede Zeichenhäufigkeit einer Tiefe zu
 let unassigned = 0
 for (let i = 0; i < charDepth.length; i++) {
  unassigned += charAmount[i]
  //solange der Pointer zu klein ist, erhöhe ihn
  while (upperLimit - (ind + charDepth[i]) > 0.5) {
   while (outer[ind] > assigned[ind]) {
    grow(outer, ind, pearls, inner, outer[ind] - assigned[ind])
   }
   while (assigned.length < inner.length) {
    assigned.push(0)
   }
   ind++
  }
  //füge die bisher unzugewiesenen Zeichen
  while (ind < outer.length && unassigned > 0) {
   let change = Math.min(unassigned, outer[ind] - assigned[ind])
   assigned[ind] += change
   unassigned -= change
   if (outer[ind] - assigned[ind] == 0) {
    ind++
   }
  }
 }
 //nähere die Anzahl an Blättern der Anzahl an verschiedenen Zeichen an
 adaptOuterSum([outer, inner], pearls, diameters, charDetail)
 return outer
}
//funktion um die Anzahl an Blättern der Anzahl an Zeichen anzugleichen
function adaptOuterSum([outer, inner], pearls, diameters, charDetail) {
 let outerSum = 0
```

```
 for (let i = 0; i < outer.length; i++) {
  outerSum += outer[i]
 }
 //berechne wie stark die Abweichung ist
 let offSet = Math.floor((outerSum - charDetail.length) / (diameters.length - 1))
 let ind = () => inner.length - 1 - (offSet < 0 ? 0 : pearls[1][pearls[1].length - 1])
 //korrigiere die Abweichung
 while (offSet != 0) {
  let neg = offSet < 0 ? 1 : -1
  let change = Math.min(Math.abs(offSet), neg > 0 ? outer[ind()] : inner[ind()])
  grow(outer, ind(), pearls, inner, neg * change)
  offSet += neg * change
 }
}
//funktion um die Codetabelle mithilfe der Blattdaten zu erstellen
function decompress(charDetail, diameters, outer) {
 let sum = 0
 for (let i = 0; i < outer.length; i++) {
  sum += outer[i]
  outer[i] -= Math.max(0, sum - charDetail.length)
  sum -= Math.max(0, sum - charDetail.length)
 }
 while (outer[outer.length - 1] == 0) {
  outer.pop()
 }
 let table = ""
 //sortiere die Zeichen absteigend nach Größe
 charDetail.sort((a, b) => b[1] - a[1])
 let charInd = 0
 //initialisiere die Liste in der die inneren Knoten der jeweiligen Tiefen gespeichert werden
 let inner = []
 for (let i = 0; i < outer.length; i++) {
  inner.push([])
 }
 inner[0].push("")
 for (let i = 0; i < outer.length; i++) {
  //füge die Codewörter für alle Zeichen dieser Tiefe hinzu
  for (let j = 0; j < outer[i]; j++) {
   table += `\n${charDetail[charInd][0]}: [${inner[i].pop()}]`
   charInd++
  }
  //Verlängere die anderen Blätter der Tiefe um die verschiedenen Perlen
  while (inner[i].length > 0) {
   let old = inner[i].pop()
   for (let j = 0; j < diameters.length; j++) {
    if (i + diameters[j] < outer.length) {
     inner[i + diameters[j]].push(old.length == 0 ? j + "" : `${old},${j}`)
    }
   }
  }
 }
 //gebe die Codetabelle zurück
 return table
}
//funktion um die Codierung für einen Text zu berechnen
function main(diamters, message) {
 //komprimiere die Perlendaten
 let pearls = compactPearls(diamters)
```

```
 //ermittle die maximale Perlengröße und entferne zu große Perlen
 let pearlLimit = heuristic(pearls, message)
 while (diamters[diamters.length - 1] > pearlLimit) {
  diamters.pop()
 }
 while (pearls[1][pearls[1].length - 1] > pearlLimit) {
  pearls[1].pop()
  pearls[0].pop()
 }
 //berechne die zusammengefassten Daten der Zeichen und deren Tiefe
 let chars = getChars(pearls, message, diamters)
 //füge die Zeichen in einen Baum ein
 let outer = greedyTree(pearls, chars, message, diamters)
 //berechne die länge des codierten Texts und dessen Codetabelle
 return [encodedLength(outer, chars), decompress(chars[3], diamters, outer), outer]
}
```