Aufgabe 1 a: Schmucknachrichten

Teilnahme-ID: 74993

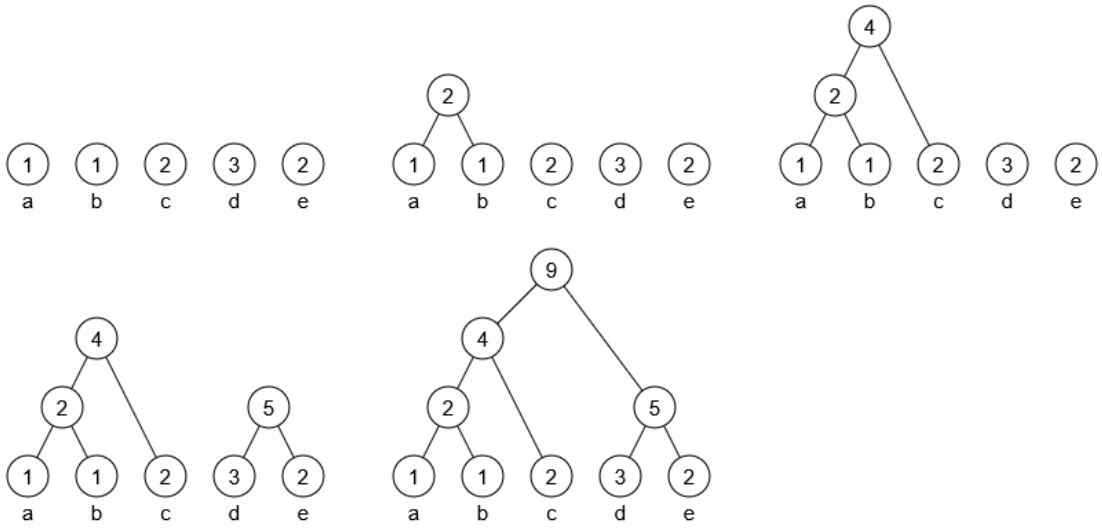
Bearbeiter dieser Aufgabe:
Mathieu de Borman

27. April 2025

Inhaltsverzeichnis

1 Lösungsidee

Sei p die Anzahl an verschiedenen Perlen. Zuerst wird gezählt, wie häufig jedes Zeichen vorkommt. Anschließend werden die Zeichen mithilfe der Huffman-Codierung codiert. Bei der Codierung wird ein Baum konstruiert, bei dem die Blätter für die zu codierenden Zeichen stehen, während der Pfad von der Wurzel zum Blatt den dazugehörigen Code darstellt. Bei dem Baum hat ein Knoten als Wert immer die Summe der Häufigkeiten seiner Kinder. Die Zeichen sind anfangs alle einknotige Teilbäume, die als Wert die eigene Häufigkeit haben. Die p kleinsten Wurzeln der Teilbäume werden dann als Kinder eines gemeinsamen, neuen Elternknotens zusammengefasst, bis es nur noch einen Baum gibt. Wie man aus zahlreicher Literatur entnehmen kann, erzeugt dieser Algorithmus eine für diese Problemstellung optimale Codierung des Texts[1]. Hier ist ein Beispiel, wie die Konstruktion eines solchen Baums für den Text "abccdddee" mit $p = 2$ aussehen würde.



[1] Huffman, D.A. (1952). „A Method for the Construction of Minimum-Redundancy Codes“, *Proceedings of the IRE*, 40(9), S. 1098–1101.

2 Umsetzung

Das Zählen der Zeichenhäufigkeiten wird mithilfe einer Hashmap umgesetzt. Anschließend werden die Häufigkeiten durch Bucketsort sortiert. Zum Konstruieren des Baums wird die 1976 von Jan van Leeuwen konzipierte Implementierung der Huffman-Codierung verwendet[2]. Diese hat den Vorteil, dass ein solcher Two Pointer leichter zu implementieren bzw. zu verstehen ist, als ein Min-Heap. Statt des Min-Heaps werden nämlich zwei Listen $leafes$ und $roots$ verwendet. $leafes$ enthält die Knoten der Zeichen in aufsteigender Reihenfolge und $roots$ immer die Wurzeln der Teilbäume in aufsteigender Reihenfolge. Anfangs ist $roots$ leer. Mit jeder Iteration werden die beiden kleinsten Elemente der jeweiligen Listen entfernt (in meiner Umsetzung wird der jeweilige Pointer auf das nächste Element verschoben), als Kinder einer neuen Wurzel zusammengefügt und zu $roots$ hinzugefügt. Dadurch, dass die Werte der beiden Kinder der neuen Wurzel größer gleich den Werten der Kinder der vorherigen Wurzel sind, wird die aufsteigende Ordnung von $roots$ sichergestellt. Am Beispiel "abccddde " mit $p = 2$:

$leafes$	1	1	2	2	3
$roots$					

$leafes$	~~1~~	~~1~~	2	2	3
$roots$	2				

$leafes$	~~1~~	~~1~~	~~2~~	2	3
$roots$	~~2~~	4			

$leafes$	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~
$roots$	~~2~~	4	5		

$leafes$	~~1~~	~~1~~	~~2~~	~~2~~	~~3~~
$roots$	~~2~~	~~4~~	~~5~~	9	

Nachdem der Baum konstruiert wurde, wird die Gesamtlänge des codierten Texts durch rekursives Iterieren über den Baum mithilfe der Funktion $encodedLength$ ermittelt. Anschließend wird mithilfe der rekursiven Funktion $createCodeTable$ eine Codetabelle erstellt.

3 Komplexitätsanalyse

Die Laufzeit- bzw. Speicherkomplexität wird in Abhängigkeit zur Länge des Eingabetexts l und der Anzahl an Perlen p formuliert. Das Zählen der Zeichenhäufigkeiten, das Sortieren der Zeichen, das Konstruieren des Baums sowie die Ermittlung der Länge des codierten Texts haben alle eine $\mathcal{O}(l + p)$ Laufzeit- und Speicherkomplexität. Das Erstellen der Codetabelle hat jedoch eine Laufzeit- und Speicherkomplexität von $\mathcal{O}(l \log l)$. Das liegt daran, dass die Codetabelle selbst $\mathcal{O}(l \log l)$ groß ist. Der Algorithmus hat also insgesamt eine Laufzeit- und Speicherkomplexität von $\mathcal{O}(l \log l + p)$

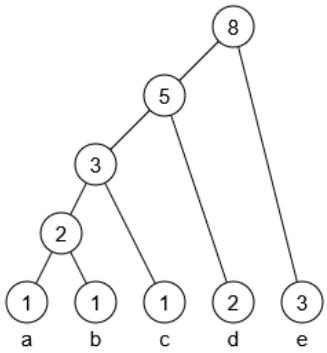
3.1 Erklärung von $\mathcal{O}(l \log l)$

Behauptung:
Die maximale Tiefe des Huffman-Baums verhält sich logarithmisch zur Länge des Eingabetexts.

Beweis:
Ich betrachte wie lang der Eingabetext mindestens sein muss, damit ein Baum die Tiefe T haben kann, um dadurch einen Rückschluss auf das asymptotische Verhalten machen zu können. Damit der Baum eine Tiefe T haben kann braucht dieser bestenfalls nur T innere Knoten mit jeweils zwei Kindern. In einem solchen Fall ist mindestens einer der beiden Kinder ein Blattknoten. Außerdem gibt es dann für jede Tiefe maximal einen inneren Knoten. Sei k alle innere Knoten und k_i der innere Knoten mit tiefe i. Sei der Wert eines Knoten g gleich $w(g)$ und dessen Tiefe gleich $t(g)$.
Im Huffman-Baum gilt aufgrund dessen Optimalität, dass für alle Knoten i und j, wenn $t(i) < t(j)$ ist, dass dann $w(i) \geq w(j)$ ist (d.h. ein höherer Knoten ist nicht kleiner als ein tieferer Knoten). Daraus folgt, dass bei einem inneren Knoten k_i mit $i < T - 2$ der Wert dessen Blattkinds b kleiner gleich $w(k_{i+2})$ ist. Da die Länge des Eingabetexts l für den T-tiefen Baum minimal sein soll, muss $w(b)$ so klein wie möglich, also gleich $w(k_{i+2})$ sein. Da $w(k_i) = w(k_{i+1}) + w(b)$ ist (d.h. der Wert des Elternteils ist gleich der Summen der Werte seiner Kinder) und außerdem $w(b) = k_{i+2}$ ist gilt $w(k_i) = w(k_{i+1}) + w(k_{t+2})$.
Dies ist nichts anderes als das rekursive Bildungsgesetz der Fibonacci-Folge. Aufgrund des asymptotischen Verhalten der Folge wissen wir, dass der Knoten k_i asymptotisch etwa das $1,6180$-Fache von k_{i+1} ist. Die Länge des Texts wächst also exponentiell mit der Höhe des Baums.

[2] Jan van Leeuwen, „On the Construction of Huffman Trees“, in: S. Michaelson und R. Milner (Hrsg.), Third International Colloquium on Automata, Languages and Programming, University of Edinburgh, July 20–23, 1976, Proceedings, Edinburgh University Press, 1976, S. 382–410.

Daraus folgt, dass die Tiefe eines Baums logarithmisch zur Textlänge wächst. Die Codelänge jedes Zeichens wächst also schlimmstenfalls Logarithmisch zur Eingabetextlänge. Wir wissen daher, dass die Codetabelle maximal $\mathcal{O}(l \log l)$ groß ist. Hier ein Beispiel für einen Baum mit $T = 4$ und minimierter Textlänge.



4 Beispiele

Wir rufen nun das JavaScript-Programm mit den verschiedenen BwInf-Eingabedateien auf. Diese Dateien sind in demselben Ordner wie die Programmdatei. Das Beispiel "schmuck02.txt" zeigt, dass das Programm auch mit Rationalen Perlenlängen umgehen kann. Das Beispiel "schmuck03.txt" zeigt, dass es nicht nur für ASCII Zeichen funktioniert. Das Programm wird mit Hilfe des Browsers ausgeführt. Mithilfe des "Datei Hochladen" Knopf kann man Beispieleingaben auswählen. Die Ausgabe erscheint auf der Webseite. Bei der Codetabelle steht immer der Index der jeweiligen Perle statt der Länge der Perle. Das Programm terminiert für alle getesteten Eingaben in weniger als 10ms auf einem gewöhnlichen PC. Für "schmuck01.txt" und "schmuck03.txt" habe ich die Codetabelle weggelassen, um diesen Abschnitt kurz zu halten. Alle Ergebnisse sind auch im Ordner der Programmdatei zu finden.

schmuck0.txt

```
Runtime: 0s
Pearllengths in mm: 1,1
Chainlength in mm: 113
S: [1,1,1]
C: [1,1,0,1,1]
H: [1,1,0,1,0]
D: [1,1,0,0]
O: [1,0,1,1]
L: [1,0,1,0]
R: [1,0,0,1]
M: [1,0,0,0]
I: [0,1,1]
N: [0,1,0]
E: [0,0,1]
 : [0,0,0]
```

schmuck00.txt

```
Runtime: 0.001s
Pearllengths in mm: 1,1,1
Messagelength in mm: 372
d: [2,2,2]
D: [2,2,1]
u: [2,2,0]
n: [2,1]
t: [2,0]
A: [1,2,2,2]
b: [1,2,2,1]
```

```
E: [1,2,2,0,1]
k: [1,2,2,0,0]
a: [1,2,1]
r: [1,2,0]
i: [1,1]
e: [1,0]
w: [0,2,2,2]
m: [0,2,2,1]
G: [0,2,2,0]
l: [0,2,1]
c: [0,2,0]
g: [0,1,2,2]
f: [0,1,2,1,2]
W: [0,1,2,1,1]
Z: [0,1,2,1,0]
P: [0,1,2,0,2]
o: [0,1,2,0,1]
?: [0,1,2,0,0]
s: [0,1,1]
h: [0,1,0]
 : [0,0]
```

schmuck01.txt

```
Runtime: 0s
Pearllengths in mm: 1,1,1,1,1
Chainlength in mm: 1150
```

schmuck02.txt

```
Runtime: 0s
Pearllengths in mm: 3.4,3.4,3.4
Chainlength in mm: 176.8
d: [2,2]
e: [2,1]
f: [2,0]
g: [1,2]
h: [1,1]
 : [1,0,2]
b: [1,0,1]
c: [1,0,0]
a: [0]
```

schmuck03.txt

```
Runtime: 0.001s
Pearllengths in mm: 2,2
Messagelength in mm: 406
```

5 Quelltext

```
//funktion um eine Liste mithilfe von Bucketsort aufsteigend zu sortieren
function bucketSort(list) {
 //finde das größte Element der Liste
 let highest = list[0][0]
 for (let i = 1; i < list.length; i++) {
  highest = Math.max(highest, list[i][0])
 }
 //erstelle die Buckets
 let buckets = []
```

```
 for (let i = 0; i <= highest; i++) {
  buckets.push([])
 }
 //Füge jedes Listenelement in den dazugehörigen Bucket ein
 for (let i = 0; i < list.length; i++) {
  buckets[list[i][0]].push(list[i])
 }
 //konvertiere die verschachtelte Liste in eine eindimensionale Liste
 let out = []
 for (let i = 0; i < buckets.length; i++) {
  out.push(...buckets[i])
 }
 return out
}
//funktion um die Summe der Längen aller Zeichen dieses Teilbaums zu ermitteln
function encodedLength(tree, diameters, depth = 0) {
 //falls es ein Blattknoten ist gebe die für dieses Zeichen benötigte länge zurück
 if (typeof tree[1] === "string") {
  return depth * tree[0]
 }
 //Andernfalls gibst du die Summe der benötigten längen deiner Nachkommen zurück
 let sum = 0
 for (let i = 1; i < tree.length; i++) {
  sum += encodedLength(tree[i], diameters, depth + diameters[tree.length - i - 1])
 }
 return sum
}
//funktion um die Codetabelle für alle Zeichen dieses Teilbaums zu erstellen
function createCodeTable(tree, code = "") {
 //falls es ein Blattknoten ist gebe den Code für dieses Zeichen zurück
 if (typeof tree[1] === "string") {
  return `\n${tree[1]}: [${code.slice(0, -1)}]`
 }
 //Andernfalls gebe die Codes für alle deine Nachkommen zurück
 let out = ""
 for (let i = 1; i < tree.length; i++) {
  out += createCodeTable(tree[i], code + (tree.length - i - 1) + ",")
 }
 return out
}
//funktion um den optimalen Huffman-Baum zu konstruieren
function main(diameters, message) {
 //Zähle wie häufig jedes Zeichen im Text 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])
  chars[i].reverse()
 }
 //berechne die Anazhl an inneren knoten
 let innerNodeAmount = Math.ceil((chars.length - 1) / (diameters.length - 1))
 //berechne wie viele Zeichen für diese Menge an inneren Knoten fehlen
 let emptyLeafes = innerNodeAmount * (diameters.length - 1) + 1 - chars.length
 //sortiere die Zeichenhäufigkeiten
 let leafes = bucketSort(chars)
```

```
 let roots = []
 let leafInd = 0
 let rootInd = 0
 const inf = Number.POSITIVE_INFINITY
 //erstelle jeden inneren Knoten
 for (let i = 0; i < innerNodeAmount; i++) {
  //erstelle einen neuen Wurzelknoten
  let newNode = [0]
  //füge für jede gegebene Perle ein Kind in den neuen Wurzelknoten ein
  for (let j = emptyLeafes; j < diameters.length; j++) {
   //wähle den kleinsten noch nicht verwendeten Teilbaum aus
   if ((roots[rootInd]||[inf])[0] > (leafes[leafInd]||[inf])[0]) {
    //füge den kleineren Teilbaum als Kind ein und verschiebe den Blattpointer
    newNode.push(leafes[leafInd])
    newNode[0] += leafes[leafInd][0]
    leafInd++
   } else {
    //füge den kleineren Teilbaum als Kind ein und verschiebe den Wurzelpointer
    newNode.push(roots[rootInd])
    newNode[0] += roots[rootInd][0]
    rootInd++
   }
   emptyLeafes = 0
  }
  roots[i] = newNode
 }
 let resultTree = roots[innerNodeAmount - 1]
 //gebe die Länge des codierten Texts und die dazugehörige Codetabelle zurück
 return [encodedLength(resultTree, diameters), createCodeTable(resultTree)]
}
```