# Aufgabe 1 b: Schmucknachrichten iterative Lösung

Teilnahme-ID: 74993

Bearbeiter dieser Aufgabe:
Mathieu de Borman

28. April 2025

## Inhaltsverzeichnis



## 1 Lösungsidee

Die beiden anderen Algorithmen (ILP und greedy Lösung), sind zwar sehr schnell, haben aber das Problem, dass diese nur Ganzzahlige Perlendurchmesser als Eingabe verwenden können. Außerdem ist die Laufzeit dieser Algorithmen sehr stark von der Größe der jeweiligen Perlen abhängig. Deswegen stelle ich eine Heuristik vor, die in den Bereichen glänzt, in denen die anderen beiden Lösungen scheitern. Die grobe Idee der Lösung ist es, zuerst irgendeine Codierung zu berechnen, und diese dann immer weiter zu verbessern. Deswegen nenne ich diese Lösung auch iterative Lösung.

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 Tiefe eines Knotens entspricht nicht der Anzahl an Kanten zwischen Wurzel und dem Knoten, sondern der größten Kante zwischen der Wurzel und dem Knoten

### 1.1 Vertauschen von Knoten

Angenommen wir haben einen beliebigen Baum und zwei verschiedenen Knoten $a$ und $b$, wobei kein Knoten Nachkomme des jeweils anderen ist. Sei $w(i)$ der Wert des Knoten $i$ und dessen Tiefe gleich $t(i)$.

Wenn ich nun die Positionen der Knoten $a$ und $b$ mitsamt Teilbaum vertausche, dann ändert sich die Codierte Textlänge um $(w(a) - w(b)) \cdot (t(b) - t(a))$. Die Idee dahinter ist, dass bei dem Tausch genau

$w(a)$ Zeichen ein um $t(b) - t(a)$ längeres Codewort erhalten. Außerdem erhalten $w(b)$ Zeichen ein um $t(a) - t(b)$ längeres Codewort. Die gesamte Änderung beträgt also:

$$w(a) \cdot (t(b) - t(a)) + w(b) \cdot (t(a) - t(b))$$
$$= w(a) \cdot (t(b) - t(a)) - w(b) \cdot (t(b) - t(a))$$
$$= (w(a) - w(b)) \cdot (t(b) - t(a))$$

Hier ein Beispiel zur Veranschaulichung:

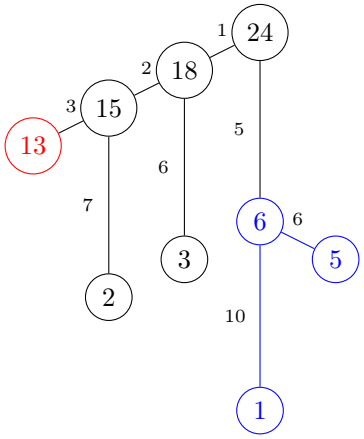


Abbildung 1: codierte Textlänge: 111 mm

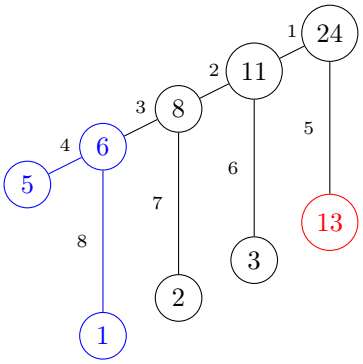


Abbildung 2: codierte Textlänge: 125 mm

## 1.2 Restrukturierung des Baums

Aus der Formel für die Vertauschung von Knoten geht hervor, dass bei einem optimalen Baum tiefere Knoten ein kleineres Gewicht haben als höhere Knoten. Die Idee des Algorithmus ist deswegen, die Knoten in absteigender Größe den jeweils höchsten inneren Positionen im Baum zuzuweisen. Dafür wird der alte Baum auf die Wurzel reduziert und die Knoten separat gespeichert, ohne, dass deren Gewichte aktualisiert werden. Außerdem bleibt gespeichert, ob der Knoten in dem Baum ein innerer oder äußerer Knoten war. Anschließend wird immer der größte noch nicht im Baum platzierte Knoten Kind des inneren Knotens des Baums, bei dem es die geringste Tiefe erhält. Das wird solange wiederholt, bis alle Knoten im Baum platziert wurden. Danach werden die Gewichte aktualisiert. Hier eine Visualisierung des Prozesses.

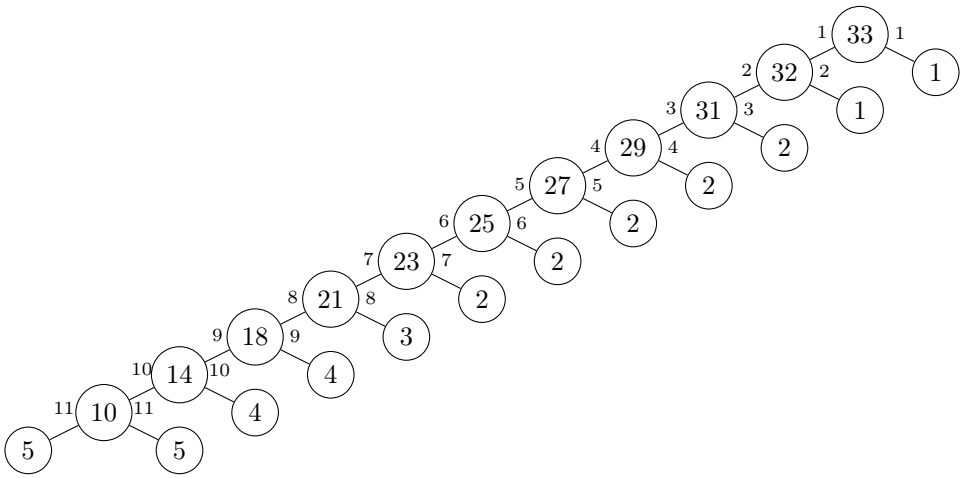


Abbildung 3: ursprünglicher Baum

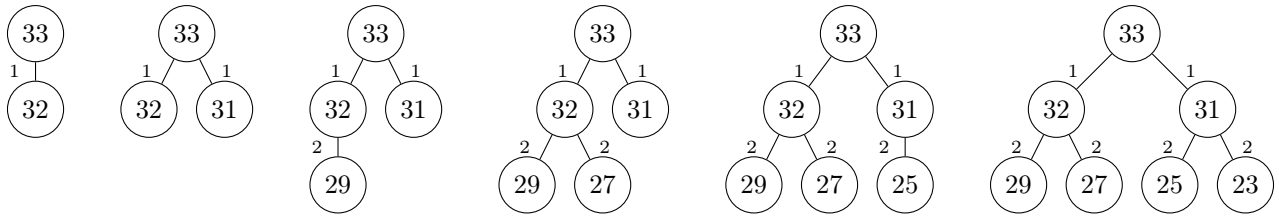


Abbildung 4: Die ersten 6 Iterationen

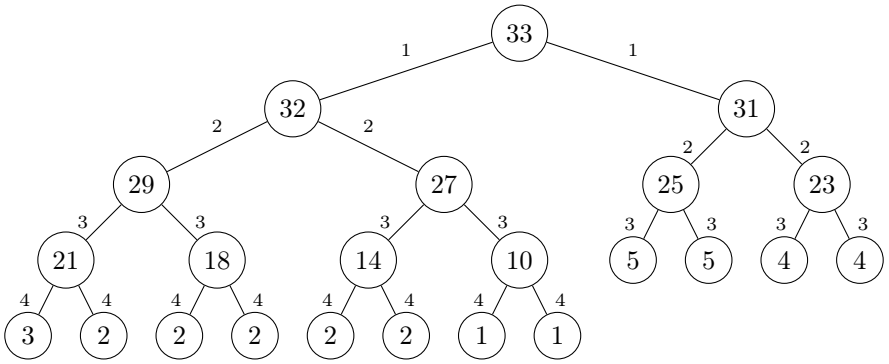


Abbildung 5: Vor dem Aktualisieren der Gewichte

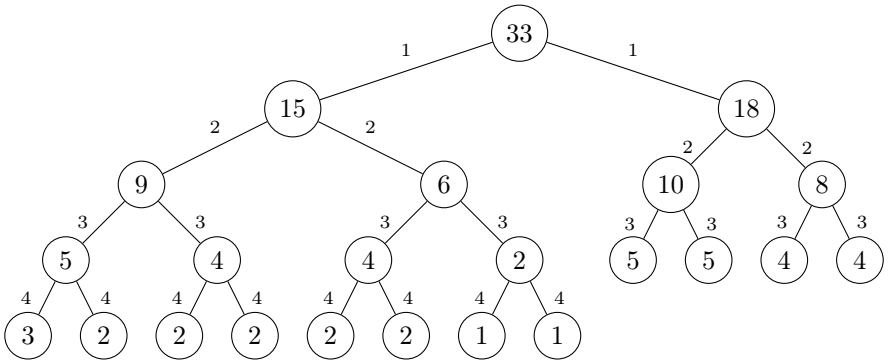


Abbildung 6: Nach dem Aktualisieren der Gewichte

Wie man an Abbildung 6 erkennen kann, ist dieser Baum noch nicht Optimal. Der Knoten mit dem Gewicht 3 könnte nämlich mit dem Knoten mit Gewicht 2 und Tiefe 3 getauscht werden. Deswegen muss ein Baum mehrmals neu zusammengebaut werden, bis dieser Algorithmus den Baum nicht weiter verbessern kann.

## 2 Umsetzung

Die Lösung wurde in JavaScript Umgesetzt. Das Zählen der Zeichenhäufigkeiten wird mithilfe einer Hashmap bewerkstelligt. Den Baum speichere ich als verschachtelte Listen. Jede Liste stellt einen Knoten dar. Der erste Wert der Liste ist die Tiefe, der Zweite das Gewicht des Knotens. Die restlichen Werte sind die Kinderknoten. Der erste Baum wird in *startTree* konstruiert. Mit *updateTree* wird dann die Länge des codierten Texts für den Baum ermittelt und der Baum aktualisiert. In *rearrange* wird der Baum umstrukturiert bzw. optimiert. Ich verwende ein selbst implementiertes Min-Heap, um zu speichern, welches die Höchsten Positionen sind, an denen ein Kind in den Baum eingefügt werden kann. Ich habe es selbst implementiert, da JavaScript keine eigene Lösung anbietet. Mithilfe von *createCodeTable* wird dann eine Codetabelle erstellt. Die Funktionen rufen sich nicht rekursiv auf, um über den Baum zu iterieren. Stattdessen wird in *updateTree* und *createCodeTable* eine Liste als Stack verwendet, um zu speichern, welche Prozesse noch erledigt werden müssen.

Das hat den Hintergrund, dass die am Anfang erstellten Bäume schnell sehr Tief werden. Wenn das Programm nun rekursiv über den Baum wandern würde, wäre es schnell an der maximalen Rekursionstiefe der Funktionen gescheitert.

## 3 Komplexitätsanalyse

Die Laufzeit- bzw. Speicherkomplexität wird in Abhängigkeit zur Länge des Eingabetexts $l$, der größten Perle $d_p$ und der Anzahl an Perlen $p$ formuliert. Anfangs werden die Perlen der Größe nach sortiert. Das hat eine Laufzeit und Speicherkomplexität von $\mathcal{O}(p \log p)$. Das Zählen der Zeichenhäufigkeiten, Erstellen des ersten Baums und Aktualisieren des Baums haben alle eine Laufzeit und Speicherkomplexität von $\mathcal{O}(l)$. Die Restrukturierung des Baums hat eine Laufzeitkomplexität von $\mathcal{O}(lp \log(lp))$ und eine Speicherkomplexität von $\mathcal{O}(lp)$. Das erstellen der Codetabelle hat eine Laufzeit und Speicherkomplexität von $\mathcal{O}(l^2)$. Ich habe keine gute obere Grenze dafür, wie oft der Baum umstrukturiert wird. Anhand der Anzahl an Wiederholungen bei den Beispieleingaben vermute ich, dass es in Richtung von $\log l$ oder $l$ sein wird, kann allerdings keine theoretische Begründung dafür angeben. Ich verwende also die maximale Textlänge des codierten Baums als obere Grenze für die Anzahl an Wiederholungen der Optimierungsfunktion. Ein Zeichen wird maximal durch einem Codewort der Länge $ld_p$ codiert. Der Ausgabetext hat dementsprechend maximal eine Länge von $l^2 d_p$. Die Laufzeitkomplexität für den Algorithmus ist insgesamt $\mathcal{O}(l^3 d_p p \log(lp)$. Die Speicherkomplexität ist $\mathcal{O}(p \log p + lp + l^2)$.

## 4 Beispiele

Wir rufen nun das JavaScript-Programm mit den verschiedenen BwInf-Eingabedateien auf. Die eigenen Beispiele "schmuck14.txt", "schmuck15.txt" und "schmuck02.txt" sollen zeigen, dass diese Lösung, im Vergleich zu den anderen beiden Lösungen für Aufgabe 1 b einen Vorteil hat. Das Programm wird mit Hilfe des Browsers ausgeführt. Mithilfe des "Datei Hochladen" Knopfs 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 vom BwInf gestellten Eingaben in weniger als 20ms auf einem gewöhnlichen PC. Für alle Eingaben habe ich die Codetabelle weggelassen, um diesen Abschnitt kurz zu halten. Die Ergebnisse von " schmuck1.txt" bis "schmuck14.txt" sind auch im Ordner der Programmdatei zu finden. Um diesen Algorithmus mit den anderen Lösungen für die Teilaufgabe 1 b vergleichen zu können, habe ich hier alle Ergebnisse tabellarisch abgebildet.

| Eingabedatei | iterative Lösung | | ILP Lösung | | greedy Lösung | |
|---|---|---|---|---|---|---|
| | Nachrichtenlänge | Laufzeit | Nachrichtenlänge | Laufzeit | Nachrichtenlänge | Laufzeit |
| schmuck0.txt | 113 | 0.002s | 113 | 0.045s | 113 | 0.001s |
| schmuck00.txt | 372 | 0.003s | 372 | 0.063s | 373 | 0.002s |
| schmuck01.txt | 1150 | 0.004s | 1150 | 0.043s | 1223 | 0s |
| schmuck02.txt | 176.8 | 0s | - | - | - | - |
| schmuck03.txt | 406 | 0.001s | 406 | 0.038s | 406 | 0s |
| schmuck1.txt | 191 | 0s | 191 | 0.047s | 191 | 0.002s |
| schmuck2.txt | 152 | 0.001s | 135 | 0.041s | 135 | 0s |
| schmuck3.txt | 279 | 0s | 279 | 0.037s | 279 | 0.002s |
| schmuck4.txt | 137 | 0s | 137 | 0.034s | 137 | 0s |
| schmuck5.txt | 3164 | 0.001s | 3162 | 0.051s | 3185 | 0.002s |
| schmuck6.txt | 234 | 0s | 234 | 0.075s | 234 | 0.001s |
| schmuck7.txt | 134563 | 0.003s | 134559 | 0.151s | 134563 | 0.038s |
| schmuck8.txt | 3302 | 0.002s | 3287 | 0.039s | 3302 | 0.001s |
| schmuck9.txt | 36642 | 0.011s | 36597 | 0.089s | 36616 | 0.002s |
| schmuck10.txt | 37422 | 0.005s | 37230 | 0.088s | 37402 | 0.031s |
| schmuck11.txt | 40239044 | 0.013s | 84886 | 0.131s | 89891 | 0.107s |
| schmuck13.txt | 7458860 | 3.165s | 7458860 | 0.442s | 7691560 | 0.274s |
| schmuck14.txt | 34455171 | 0.083s | 34425926 | 7.903s | 34484070 | 0.095s |
| schmuck15.txt | 1194153268 | 0.085s | - | >60s | 1205177074 | 15.495s |

## 5 Quelltext

```
//funktion um den ersten Baum zu berechnen
function startTree(diameters, chars) {
 //erstelle den Wurzelknoten
 let depth = 0
 let root = [depth, 0]
 let nodes = [root]
 let growingNode = root
 //füge immer bei dem tiefsten Knoten weitere Kinder ein
 while (chars.length > 0) {
  for (let i = 0; i < diameters.length - 1 && chars.length > 0; i++) {
   growingNode.push([depth + diameters[i], ...chars.pop()])
   nodes.push(growingNode[i + 2])
  }
  depth += diameters[growingNode.length - 2]
  if (chars.length == 1) {
   growingNode.push([depth, ...chars.pop()])
   nodes.push(growingNode[growingNode.length - 1])
  } else if (chars.length > 0) {
   growingNode.push([depth, 0])
   nodes.push(growingNode[growingNode.length - 1])
   growingNode = growingNode[growingNode.length - 1]
  }
 }
 return [root, nodes]
}
//aktualisiere die Werte für den Baum
function updateTree(diameters, root, nodes) {
 //setze alle Gewichte erstmal auf unberechnet
 let score = 0
 for (let i = 1; i < nodes.length; i++) {
  if (typeof nodes[i][2] !== "string") {
   nodes[i][1] = 0
  }
 }
 let stack = [root]
 //solange noch nicht alle Knoten aktualisiert sind
 while (stack.length > 0) {
  //schaue, ob schon alle Kinder aktualisiert sind
  let node = stack.pop()
  let childsUpdated = 1
  for (let i = 2; i < node.length; i++) {
   childsUpdated *= node[i][1]
   node[i][0] = node[0] + diameters[i - 2]
  }
  //falls die Kinder aktualisiert sind, wird das eigene Gewicht aktualisiert
  if (childsUpdated) {
   let sum = 0
   for (let i = 2; i < node.length; i++) {
    sum += node[i][1]
    if (typeof node[i][2] === "string") {
     score += node[i][0] * node[i][1]
    }
   }
   node[1] = sum
   //falls die Kinder noch nicht aktualisiert sind, müssen diese noch aktualisiert werden
  } else {
```

```
   stack.push(node)
   for (let i = 2; i < node.length; i++) {
    if (!node[i][1]) {
     stack.push(node[i])
    }
   }
  }
 }
 //gebe die codierte Textlänge aus
 return score
}
//funktion um den Baum umzustrukuturieren
function rearrange(root, nodes, diameters) {
 //zerlege den Baum
 for (let i = 0; i < nodes.length; i++) {
  if (typeof nodes[i][2] !== "string") {
   nodes[i].splice(2)
  }
 }
 //sortiere die Knoten nach Gewicht
 nodes.sort((a, b) => b[1] - a[1])
 let heap = []
 //füge alle möglichen Kinderplätze in den heap ein
 for (let i = 0; i < diameters.length; i++) {
  heapInsert([root, diameters[i]], heap)
 }
 //füge jeden Knoten in den Baum ein
 for (let i = 0; i < nodes.length; i++) {
  if (nodes[i] != root) {
   let [parent, depth] = heapPopMin(heap)
   nodes[i][0] = depth
   parent.push(nodes[i])
   //falls der Knoten ein innerer Knoten ist, dass entstehen neue Kinderplätze
   if (typeof nodes[i][2] !== "string") {
    for (let j = 0; j < diameters.length; j++) {
     heapInsert([nodes[i], depth + diameters[j]], heap)
    }
   }
  }
 }
 //aktualisiere die Gewichte
 return updateTree(diameters, root, nodes)
}
//erstelle die Codetabelle
function createCodeTable(root) {
 let table = ""
 let stack = [[root, []]]
 //solange noch nicht alle Zeichen einen Code bekommen haben
 while (stack.length > 0) {
  let [node, code] = stack.pop()
  //falls es ein Blattknoten ist, füge das Codewort in die Tabelle ein
  if (typeof node[2] === "string") {
   table += `\n${node[2]}: [${code}]`
  }
  //falls es ein innerer Knoten ist, müssen Codewörter für alle Kinder erstellt werden
  for (let i = 2; i < node.length; i++) {
   stack.push([node[i], [...code, i - 2]])
  }
```

```
 }
 return table
}
//funktion um eine Codierung mithilfe von der Restrukturierung von Bäumen zu ermitteln
function main(diameters, message) {
 //sortiere die Durchmesser
 diameters.sort((a, b) => a - b)
 //Zähle die Zeichenhäufigkeiten
 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()
 }
 chars.sort((a, b) => b[0] - a[0])
 //erstelle einen ersten Baum
 let [root, nodes] = startTree(diameters, [...chars])
 let last = Number.POSITIVE_INFINITY
 //verbessere den Baum solange, bis er nicht besser wird
 let recent = updateTree(diameters, root, nodes)
 while (recent < last) {
  last = recent
  recent = rearrange(root, nodes, diameters)
 }
 //gebe die Codetbelle und die Textlänge des codierten Texts zurück
 return [last, createCodeTable(root)]
}
```