

## Übungsblatt mit Lösungen 12

### Aufgabe T41

Gegeben ist ein Baum mit  $n$  Knoten. Jeder Knoten  $v$  hat ein nicht-negatives Gewicht  $w(v)$ . Entwerfen Sie einen effizienten Algorithmus, welcher eine unabhängige Knotenmenge (paarweise keine Kanten, auch Independent Set genannt) mit maximalem Gewicht findet.

Das Gewicht einer Knotenmenge ist die Summe der Gewichte der einzelnen Knoten.

Wie schnell ist Ihr Algorithmus und wieviel Speicher benötigt er?

### Lösungsvorschlag

Wir wollen das Problem mittels dynamischer Programmierung lösen. Sei der Baum  $T$  und die Wurzel  $r$ . Wir bezeichnen als  $T(v)$  den Teilbaum, der  $v$  als Wurzel hat.  $T(r)$  ist demnach der gesamte Baum. Sei  $A(v)$  die optimale Lösung in  $T(v)$  und  $B(v)$  die optimale Lösung in  $T(v) - \{v\}$  (die optimale Lösung wenn  $v$  nicht Teil der Lösung ist). Die Lösung des Problems ist der Wert  $A(r)$ , den wir berechnen wollen.

Wir beginnen bei den Blättern  $l$ :  $A(l) = w(l)$  und  $B(l) = 0$ . Sei im folgenden  $v$  ein Knoten mit  $m$  Kindern  $u_1, \dots, u_m$ .

Die optimale Lösung in  $T(v)$  ist nun die bessere der folgenden beiden Möglichkeiten:

- $v$  nicht hinzuzufügen.
- $v$  hinzuzufügen.

Falls wir  $v$  nicht wählen (Wert  $B(v)$ ), können wir einfach die optimalen Lösungen der Teilbäume der Kinder zusammenfügen, d.h.  $B(v) = A(u_1) + \dots + A(u_m)$ .

Im zweiten Fall darf man keines der Kinder von  $v$  hinzufügen, da sie eine Kante zu  $v$  besitzen. Deshalb müssen wir die  $B$  Werte der Kinder wählen, und dazu das Gewicht von  $v$  addieren. Die bessere der beiden Möglichkeiten finden wir mit der maximums funktion, d.h.  $A(v) = \max\{B(v), w(v) + B(u_1) + \dots + B(u_m)\}$ .

Da jeder Knoten nur einmal durchlaufen wird und der Zeitaufwand in jedem Knoten Konstant ist beträgt die Laufzeit  $O(n)$ . Der Speicheraufwand ist ebenfalls  $O(n)$  da für jeden Knoten nur die zwei Werte  $A(v)$  und  $B(v)$  gespeichert werden müssen.

### Aufgabe T42

Gegeben ist ein sehr langer Text  $w$  und sehr viele kurze Wörter  $u_1, \dots, u_n$ . Ziel ist es herauszufinden, wie oft jedes der Wörter  $u_i$  im Text  $w$  vorkommt.

Entwerfen Sie einen effizienten Algorithmus, der diese Aufgabe in einer Laufzeit von  $O((|w| + |u_1| + \dots + |u_n|) \log(|w|))$  Schritten löst.

## Lösungsvorschlag

Erstelle zuerst das Suffix-Array für  $w$ . Dann gilt Folgendes:

Die Idee ist, dass es für jedes Vorkommen von  $u$  in  $v$  genau ein Suffix gibt, das mit  $u$  beginnt. Alle Suffixe, die mit  $u$  beginnen, stehen aber in der lexikographischen Sortierung hintereinander und folgen deshalb im Suffix-Array aufeinander. Wenn wir also die erste und die letzte Stelle finden, an der ein mit  $u$  beginnendes Suffix steht, können wir deren Anzahl ganz einfach durch Subtraktion ermitteln.

Der Algorithmus sieht also wie folgt aus: Durchsuche das Suffix-Array mit binärer Suche nach der ersten Stelle, an der ein mit  $u$  beginnendes Suffix steht. Das geht, indem man bei Suffixen, die mit einem lexikographisch größeren String als  $u$  beginnen, weiter vorne sucht, und sonst weiter hinten. Die Suche nach der letzten Stelle, an der ein mit  $u$  beginnendes Suffix steht, verläuft analog. Wenn nun Stelle  $i$  im Suffix-Array die erste Stelle ist, an der ein mit  $u$  beginnendes Suffix steht, und Stelle  $j$  die letzte solche Stelle, dann haben wir  $j - i + 1$  Vorkommen von  $u$  in  $v$ .

Zur Laufzeit: Da wir ein Array der Länge  $|v|$  durchsuchen müssen, brauchen wir für die binäre Suche  $O(\log |v|)$  Schritte. Und da wir in jedem Schritt das Präfix des Suffixes mit  $u$  vergleichen müssen, brauchen wir für jeden Schritt  $O(|u|)$  Schritte. Insgesamt ergibt sich also die geforderte Laufzeit in  $O(|u| \cdot \log |v|)$ .

## Aufgabe T43

Gegeben sei ein Text, den wir uns als von Leerzeichen und Zeilenumbrüchen getrennte Folge von Wörtern vorstellen. Der Text soll nun linksbündig auf einer Schreibmaschine getippt werden. (Auf einer Schreibmaschine hat jeder Buchstabe die gleiche Breite, wie man es auch von Texteditoren kennt.) Dabei sollen aber keine Wörter getrennt werden, sodass am rechten Seitenrand mit Leerzeichen aufgefüllt werden muss. Der Seitenrand befindet sich nach der 75. Position. Die ideale Länge einer Zeile ist also auch 75. Ist eine Zeile aber am rechten Rand mit  $k$  Leerzeichen aufgefüllt, dann ist ihre Hässlichkeit genau  $k \cdot k$  und die Hässlichkeit eines gesetzten Textes ist die Summe der Hässlichkeiten aller Zeilen. Entwerfen Sie einen Algorithmus, der die Wörter so auf die Zeilen verteilt, dass die Hässlichkeit minimal ist. Wie könnte man das Problem alternativ lösen?

Die Hässlichkeit des obigen Textes ist 187 und besser geht es auch nicht.

## Lösungsvorschlag

Musterlösung in C++ s. para.cpp in sciebo. Hier eine Erläuterung der dynamischen Programmierung:

Äußere Schleife (`for (int i=0; i < (int)words.size(); i++)`): Iteriert über jedes Wort und berechnet die minimale Badness, wenn das Formatieren bis zum Wort  $i$  endet.

`penalty`: Initialisiert mit einem großen Wert, um die minimale Badness für die aktuelle Zeile zu finden.

`linebreak`: Variable, um den Index des vorherigen Zeilenumbruchs zu speichern.

Innere Schleife (`for (int j=i-1; j >= -1; j--)`): Versucht, einen Zeilenumbruch vor jedem Wort  $j$  zu setzen und berechnet die entsprechende Badness.

`p1`: Badness für das Formatieren bis Wort  $j$ .

`p2`: Badness für die aktuelle Zeile von  $j+1$  bis  $i$  (`p1 + len[i] - len[j] - 1`).

Badness-Funktion (`p(int l)`): Berechnet die Badness für eine Zeile der Länge  $l$ . Wenn die Länge größer als 75 ist, gibt sie einen hohen Wert zurück (10000000), andernfalls das Quadrat der Differenz zu 75.

Wenn die Summe  $p_1 + p_2$  kleiner als die aktuelle penalty ist, wird penalty aktualisiert und linebreak gesetzt.

Wenn  $p_2$  größer als 10000 ist, bricht die Schleife ab, um unnötige Berechnungen zu vermeiden.

### Aufgabe H38 (10 Punkte)

Konstruieren Sie einen optimalen Suchbaum für die Schlüssel  $A$ ,  $B$ ,  $C$  und  $D$ . Auf diese wird mit den Wahrscheinlichkeiten 0.2, 0.3, 0.1 und 0.4 zugegriffen.

Erstellen Sie dazu die Tabellen für  $w_{i,j}$  und  $e_{i,j}$ .

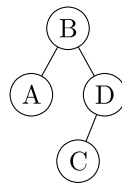
### Lösungsvorschlag

Die Tabellen sehen wie folgt aus:

$w_{i,j}$	$A$	$B$	$C$	$D$
A	0.2	0.5	0.6	1.0
B	0.0	0.3	0.4	0.8
C	0.0	0.0	0.1	0.5
D	0.0	0.0	0.0	0.4

$e_{i,j}$	$A$	$B$	$C$	$D$
A	0.2(A)	0.7(B)	0.9(B)	1.8(B)
B	0.0	0.3(B)	0.5(B)	1.3(D)
C	0.0	0.0	0.1(C)	0.6(D)
D	0.0	0.0	0.0	0.4(D)

Daraus entsteht folgender optimaler Suchbaum:



**Genauere Erklärung:** Die erste Tabelle enthält in Zeile  $i$ , Spalte  $j$  den Wert für  $w_{i,j}$ , was als  $\sum_{k=i}^j p_k$  in der Vorlesung definiert wurde. Das heißt, ein Baum, der die Schlüssel  $i$  bis  $j$  enthält, wird im Erwartungswert bei einer Suche nach einem zufälligen Schlüssel  $w_{i,j}$  mal besucht.

In der Vorlesung wurde präsentiert, wie man die Werte dieser ersten Tabelle effizienter mit Hilfe von dynamischer Programmierung berechnen kann. Hier wird einfach nur ausgenutzt, dass wenn man den Wert für  $w_{i,j-1}$  hat,  $w_{i,j} = w_{i,j-1} + p_j$  ist. Dies folgt direkt aus der Definition von  $w_{i,j}$ .

Wir wollen jetzt mit Hilfe dieser Tabelle einen optimalen Suchbaum berechnen. Wir definieren  $e_{i,j}$  als den Erwartungswert der Anzahl der Vergleiche in einem optimalen Suchbaum, der die Schlüssel von  $i$  bis  $j$  enthält. Sei  $n$  die Anzahl der Schlüssel. Wir wollen den Baum finden, für den  $e_{1,n}$  minimal ist, da  $e_{1,n}$  die Anzahl der erwarteten Vergleiche in einem Baum, der alle Schlüssel enthält, ist.

Es gilt  $e_{i,j} = \min_{i \leq r \leq j} (e_{i,r-1} + e_{r+1,j}) + w_{i,j}$ : Die Summe  $(e_{i,r-1} + e_{r+1,j}) + w_{i,j}$  drückt für ein bestimmtes  $r$  aus, dass die erwartete Anzahl an Vergleichen für einen optimalen Suchbaum, der die Schlüssel von  $i$  bis  $j$  enthält, wobei der Schlüssel  $r$  der Wurzelknoten ist, die Summe folgender Teile ist:

- erwartete Anzahl für die Besuche des Wurzelknotens  $r$  ( $w_{i,j}$ )
- optimale erwartete Anzahl an Vergleichen im linken Unterbaum ( $e_{i,r-1}$ )
- optimale erwartete Anzahl an Vergleichen im rechten Unterbaum ( $e_{r+1,j}$ )

Es ist wichtig hier anzumerken, dass wenn  $j < i$  ist,  $e_{i,j} = 0$  ist, da dies ein leerer Unterbaum ist. Wenn wir das Minimum über alle möglichen Wurzelknoten wählen, ist das Ergebnis der optimale Wert  $e_{i,j}$ .

Wir berechnen  $e_{1,n}$  wie folgt: Der optimale Suchbaum mit einem einzigen Knoten ist der Knoten selbst. Es gilt also, dass  $e_{i,i} = w_{i,i}$ , da ein Baum mit einem Knoten im Erwartungswert genau so oft besucht wird wie der Knoten im Erwartungswert besucht wird. Jetzt können wir mit Hilfe der Formel  $e_{i,j} = \min_{i \leq r \leq j} (e_{i,r-1} + e_{r+1,j}) + w_{i,j}$  alle Werte berechnen, wenn  $|i - j| = 2$  ist, also für Bäume, die zwei Schlüssel enthalten, da wir alle Werte  $e_{i,j}$  kennen, wenn  $i = j$  ist.

Wir können weiterhin die Werte für Bäume mit immer mehr Knoten berechnen, da wir in der Formel nur die Werte für Bäume brauchen, die weniger Schlüssel enthalten. Jedesmal können wir auch in der Tabelle vermerken, welchen Knoten wir als Wurzelknoten gewählt haben, als wir das Minimum genommen haben. Letztendlich werden wir den optimalen Wert für alle Schlüssel  $e_{1,n}$  berechnen.

Wir können jetzt aus der Tabelle den optimalen Suchbaum auslesen: In der Zelle für den Wert  $e_{1,n}$  steht die Wurzel  $r$  des Baums. Da er ein Suchbaum ist, wissen wir, dass der linke Unterbaum die Schlüssel von 1 bis  $r - 1$  enthält, und der rechte Unterbaum die Schlüssel von  $r + 1$  bis  $n$ . In der Tabelle finden wir dann in der Zelle für  $e_{1,r-1}$  die Wurzel vom linken Unterbaum und in  $e_{r+1,n}$  die Wurzel vom rechten Unterbaum. Wir können nach dieser Logik dann den Baum rekursiv aufbauen, indem wir immer in der Tabelle nachgucken, was die nächsten Wurzeln sind. Das Ergebnis ist dann der optimale Suchbaum.

### Aufgabe H39 (20 Punkte)

Sie möchten in einer längeren Zeichenkette  $w$  ein Wort  $u$  finden, so dass  $uuu$  ein Unterwort von  $w$  bildet. Insbesondere interessiert es Sie, wie lang so ein  $u$  maximal sein kann.

Entwerfen Sie einen halbwegs effizienten Algorithmus, der dieses Problem lösen kann und implementieren Sie ihn in einer vernünftigen Programmiersprache. Ihr Verfahren sollte so effizient sein, dass es mit  $|w| = 50.000$  fertig wird, ohne dass man allzu lange warten muss.

Erläutern Sie Ihre Vorgehensweise und reichen Sie Ihr Programm ein. Im Moodle-Lernraum finden sie die drei Dateien `input1.txt`, `input2.txt` und `input3.txt`. Jede umfasst zehn Beispieleingaben in den ersten zehn Zeilen, d.h. in jeder dieser Zeilen steht ein  $w$ . Lösen Sie diese Beispiele. Für die volle Punktzahl sollten Sie wenigstens die ersten beiden Beispieleingaben vollständig lösen. Geben Sie die entsprechenden maximalen Längen Ihrer Lösungen an.

### Lösungsvorschlag

Die richtigen Lösungen sind:

193, 156, 235, 191, 187, 205, 141, 177, 232, 219

1043, 631, 640, 854, 1226, 1035, 1110, 735, 770, 1085

42718, 25673, 30823, 44696, 39961, 35557, 28194, 46268, 45976, 32325

TODO: Peters Implementierung einfügen

### Aufgabe H40 (10 Punkte)

Entwerfen Sie einen Algorithmus, der die Swap-Edit-Distance zwischen zwei gegebenen Strings  $u$  und  $v$  berechnet. Diese ist definiert als die minimale Anzahl von Swap-Edit-Operationen, die man benötigt, um aus dem String  $u$  den String  $v$  zu machen. Es gibt drei Swap-Edit-Operationen:

1. Einfügen: Man kann an einer beliebigen Stelle einen Buchstaben einfügen.
2. Löschen: Man kann an einer beliebigen Stelle einen Buchstaben löschen.
3. Tauschen: Man kann zwei benachbarte Buchstaben tauschen.

Beispiel: Die Swap-Edit-Distance von  $u = abba$  und  $v = ababb$  beträgt 2, etwa indem man zuerst die beiden letzten Buchstaben von  $u$  vertauscht und dann am Ende ein  $b$  einfügt.

## Lösungsvorschlag

Lösung mittels dynamischer Programmierung. Wir berechnen die Swap-Edit-Distance von Paaren  $u, v$  mit aufsteigendem  $|u| + |v|$ . Um nun auszunutzen, dass für (insgesamt kürzere) Präfixe  $u'$  und  $v'$  die Swap-Edit-Distance bereits berechnet worden ist, kann man bei der Berechnung der Swap-Edit-Distance der (insgesamt längeren Wörter)  $u$  und  $v$  auf diese zurückgreifen. Dazu schaut man verschiedene Möglichkeiten an, um  $v$  aus  $u$  zu machen, und wählt dann die beste, also die mit den wenigsten Operationen.

1. Wir können den letzten Buchstaben aus  $u$  entfernen, und das entstehende  $u'$  zu  $v$  transformieren. Die Swap-Edit-Distance zwischen  $u'$  und  $v$  ist bereits bekannt.
2. Wir können den letzten Buchstaben aus  $v$  entfernen, und  $u$  zum entstandenen  $v'$  transformieren. Die Swap-Edit-Distance zwischen  $u$  und  $v'$  ist bereits bekannt.
3. Wir können die letzten Buchstaben aus  $u$  und aus  $v$  entfernen, und das entstehende  $u'$  zum entstehenden  $v'$  transformieren. Die Swap-Edit-Distance zwischen  $u'$  und  $v'$  ist bereits bekannt.
4. (Zu einem gegebenen Wort  $w$  sei  $w_{\leftrightarrow}$  das Wort  $w$  mit vertauschten Endbuchstaben.) Wir können bei den obigen drei Möglichkeiten auch jeweils die letzten beiden Buchstaben vertauschen, sodass wir nicht nur die Swap-Edit-Distance von  $u^{(l)}$  und  $v^{(l)}$  in Betracht ziehen, sondern auch die von  $u_{\leftrightarrow}^{(l)}$  und  $v_{\leftrightarrow}^{(l)}$ ,  $u_{\leftrightarrow}^{(l)}$  und  $v^{(l)}$  sowie  $u^{(l)}$  und  $v_{\leftrightarrow}^{(l)}$ .