Ludwig-Maximilians-Universität München
Institut für Informatik
Prof. Dr. Thomas Seidl
Janina Sontheim, Florian Richter

München, 23.07.2019

Algorithmen und Datenstrukturen

SS 2019

Übungsblatt 13: Paradigmen II

Aufgabe 13-1 *Dynamische Programmierung*

(a) Berechnen Sie die Edit-Distanz der Wörter `ALGORITHMEN` und `DATENSTRUKTUREN` mit Hilfe dynamischer Programmierung.

(b) Geben Sie die Komplexität der dynamischen Programmierung für dieses Beispiel im Gegensatz zu einer naiven rekursiven Lösung zur Berechnung der Edit-Distanz an.

Lösungsvorschlag:

(a)

		A	L	G	O	R	I	T	H	M	E	N
	0	1	2	3	4	5	6	7	8	9	10	11
D	1	**1**	2	3	4	5	6	7	8	9	10	11
A	2	**1**	2	3	4	5	6	7	8	9	10	11
T	3	**2**	2	3	4	5	6	6	7	8	9	10
E	4	**3**	3	3	4	5	6	7	7	8	8	9
N	5	4	**4**	4	4	5	6	7	8	8	9	8
S	6	5	5	**5**	5	5	6	7	8	9	9	9
T	7	6	6	6	**6**	6	6	6	7	8	9	10
R	8	7	7	7	7	**6**	7	7	7	8	9	10
U	9	8	8	8	8	**7**	7	8	8	8	9	10
K	10	9	9	9	9	8	**8**	8	9	9	9	10
T	11	10	10	10	10	9	9	**8**	9	10	10	10
U	12	11	11	11	11	10	10	9	**9**	10	11	11
R	13	12	12	12	12	11	11	10	10	**10**	11	12
E	14	13	13	13	13	12	12	11	11	11	**10**	11
N	15	14	14	14	14	13	13	12	12	12	11	**10**

(b) Dynamische Programmierung:
Wie in der Vorlesung gezeigt, liegt die Komplexität für zwei Zeichenketten s und t bei $O(|s| \cdot |t|)$. Für unser Beispiel müssen demnach maximal ca. $11 \cdot 15 = 165$ Operationen bei dynamischer Programmierung durchgeführt werden.

Rekursive Lösung:
Für die rekursive Lösung müssen im schlechtesten Fall in jeder Iteration 3 rekursive Aufrufe getätigt werden, um das Minimum dieser Aufrufe berechnen zu können. Dabei verringert sich immer die Länge mindestens einer der Zeichenketten s oder t um 1. D.h. können maximal $|s| + |t|$ Rekursionsschritte gemacht werden. Insgesamt ergibt sich damit also eine obere Schranke für die Laufzeit von $O(3^{|s|+|t|})$. Das bedeutet für unser Beispiel eine Durchführung von maximal $3^{11+15} = 3^{26} \approx 2.5 \cdot 10^{12}$

Aufgabe 13-2 *Greedy Algorithmen, Heuristiken und dynamische Programmierung*

In der Vorlesung haben Sie kennengelernt, dass Greedy Algorithmen, welche auf Heuristiken basieren, oft nicht zur optimalen Lösung von Problemen führen. Dennoch finden sich in der Industrie sehr viele Beispiele, bei denen Greedy Algorithmen zur Approximation einer optimalen Lösung eingesetzt werden.

(a) Greedy Algorithmen können die optimale Lösung finden, tuen es aber nicht immer. Warum werden überhaupt Greedy Algorithmen verwendet und nicht stattdessen Algorithmen, welche eine optimale Lösung garantieren?

(b) Das Rucksackproblem gehört zur Liste der 21 klassischen NP-vollständigen Problemen. Um eine optimale Lösung für dieses Problem finden zu können, gibt es bisher nur den Lösungsansatz alle Möglichkeiten systematisch durchzuprobieren.

(i) Welche Komplexität ergibt sich demnach für den Algorithmus, welcher die optimale Lösung des Rucksackproblems mit n verschiedenen Gegenständen liefert.

(ii) Wie groß ist die Komplexität, wenn eine optimale Lösung mit Hilfe von der in der Vorlesung besprochenen dynamischen Programmierung für das Rucksackproblem mit n verschiedenen Gegenständen und einer Kapazität von b berechnet wird.

(iii) Warum ist das Ergebnis von (ii) kein Widerspruch zu (i)?

Lösungsvorschlag:

(a) Weil Greedy Algorithmen meist mit Heuristiken arbeiten, resultiert durch deren Ausführung meistens eine Lösung, die zwar nicht optimal, jedoch im Durchschnitt sehr gut ist. Mit Hilfe der Heuristiken ist zudem sichergestellt, dass Greedy Algorithmen keine allzu große Komplexität haben. Gerade für Probleme, welche eine hohe Komplexität zur Berechnung ihrer optimalen Lösung haben (beipielsweise NP-vollständige Probleme wie Traveling Salesman Problem), bieten sich demnach Greedy Algorithmen an. So wird sichergestellt, dass in jedem Fall eine mögliche und vermutlich auch eine sehr gute, wenn auch nicht die beste Lösung, erreicht wird.

(b) (i) Es kann eine Menge mit allen kombinatorischen Möglichkeiten an Gegenständen gebildet werden, welche in dem Rucksack enthalten sein könnten. Für eine Menge N von verschiedenen Gegenständen mit $|N| = n$ entspricht das genau der Menge all ihrer Teilmengen also ihrer Potenzmenge $\mathcal{P}(N)$.
Der Betrag dieser Menge ist also die Anzahl aller möglichen Lösungen $|\mathcal{P}| = 2^{|N|} = 2^n$ und damit gilt für die Komplexität: $O(2^n)$.

(ii) Für die doppelte for-Schleife benötigt der Algorithmus aus der Vorlesung in einer Iteration $n \cdot b$ Durchgänge. Innerhalb der Schleifen können alle Rechenoperationen in konstanter Zeit $O(1)$ durchgeführt werden. Damit ergibt sich als Gesamtlaufzeit $O(n \cdot b)$.

(iii) Die Laufzeit scheint nun linear zu n und b, und damit wesentlich effizienter zu sein als die in (i) berechnete Komplexität von $O(2^n)$. Dennoch ist dies nur scheinbar so, denn b ist zu seiner Eingabelänge exponentiell wachsend, sodass die Komplexität nur pseudopolynomiell ist. Denn die Durchführung des Algorithmus hängt genauer nicht von b direkt, sondern von dessen Länge ab. Unter der Annahme einer sinnvollen Kodierung ist die Länge von b logarithmisch zu b. D.h. für eine Erhöhung von b um m steigt die Laufzeit für b nicht linear auf $b + m$, sondern auf $2^{\log(b+m)}$. Also gilt für die Komplexität von b das exponentiell zu dessen Eingabelänge wächst: $O(2^{\log(b)})$. Für die gesamte Komplexität gilt also $O(n \cdot 2^{\log(b)}) = O(2^{\log(n)} \cdot 2^{\log(b)}) = O(2^{\log(n)+\log(b)}) = O(2^{\log(n \cdot b)})$. Das entspricht schließlich $O(2^n)$. TODO Lösung verbessern!