Aufgabensammlung ADS-Repetitorium WS 24/25

Dynamisches Programmieren

Aufgabe 1: SubsetSum

Gegeben sind eine Liste $A = \langle o_1, o_2, \ldots, o_n \rangle$ von n natürlichen Zahlen sowie ein $b \in \mathbb{N}$. Gefragt ist, ob die Summe von einem Teil der in A enthaltenen Zahlen genau b ergibt. Formal ist also gefragt, ob ein $I \subseteq \{1, 2 \ldots, n\}$ mit $\sum_{i \in I} o_i = b$ existiert. Ein naiver Ansatz wäre es, alle Teilmengen der in A enthaltenen Zahlen auszuprobieren. Da es jedoch 2^n viele solche Teilmengen gibt, hat dieser Ansatz eine Laufzeit von $\Omega(2^n)$. Wir wollen stattdessen ein dynamisches Programm angeben, dessen Laufzeit in $\mathcal{O}(n \cdot b)$ liegt.

(a) Wie sehen geeignete Teilprobleme aus, die hierfür gelöst werden müssen?

Lösung: Für $1 \leq i \leq n$ und $0 \leq \tilde{b} \leq b$ definieren wir das Teilproblem $T[i, \tilde{b}]$:

$$T[i, \tilde{b}] := \begin{cases} 1 & \text{falls mit einem Teil der Objekte } o_1, \ldots, o_i \text{ die Summe } \tilde{b} \text{ erreicht werden kann} \\ 0 & \text{sonst} \end{cases}$$

(b) Wir suchen nun zunächst eine rekursive Formulierung, mit der die Antwort auf die Teilprobleme berechnet werden kann. Welchem Teilproblem entspricht die Lösung der eigentlichen Aufgabe?

Lösung: Für $i = 1$ definieren wir den Basisfall.

$$T[1, \tilde{b}] := \begin{cases} 1 & \text{falls } o_1 = \tilde{b} \text{ oder } \tilde{b} = 0 \\ 0 & \text{sonst} \end{cases}$$

Dann können wir für ein $i > 1$ die Fallunterscheidung verwenden, ob Objekt o_i in der Lösung verwendet werden soll, oder nicht. Jeweils muss dann die restliche Lösung aus den Objekten $o_1, \ldots, o_{i-1}$ gebildet werden.

$$T[i, \tilde{b}] := \max \Big\{ T[i-1, \tilde{b} - o_i], T[i-1, \tilde{b}] \Big\}$$

Die gesuchte Antwort entspricht dem Teilproblem $T[n, b]$.

Hinweis: Für bessere Lesbarkeit ignorieren wir im rekursiven Schritt den Fall $\tilde{b} < o_i$, für den das linke Teilproblem nicht definiert ist. Wir nehmen hier und im Folgenden stillschweigend an, dass für solche undefinierten Werte Standardmäßig 0 als Antwort gegeben wird.

(c) Nun soll diese rekursive Lösung in einem dynamischen Programm berechnet werden. Wie groß ist der Speicherbedarf dieser Lösung?

Lösung:

Algorithmus 1: SubsetSum(int[ ] A, b)

```
1  Sei T eine Tabelle der Größe n × (b + 1)
2  for b̃ = 0 to b do
3      if b̃ = 0 or b̃ = A[1] then
4          T[1, b̃] = 1
5      else
6          T[1, b̃] = 0
7  for i = 2 to n do
8      for b̃ = 0 to b do
9          T[i, b̃] = max{T[i − 1, b̃ − A[i]], T[i − 1, b̃]}
10 return T[n, b]
```

(d) Ist es möglich, einen von n unabhängigen Speicherbedarf zu erreichen? Wenn ja, wie lässt sich das erreichen?

Lösung: Wir können beobachten, dass immer nur auf die vorhergehende Spalte in der Tabelle zugegriffen wird. Daher können wir statt der $n \times (b+1)$ Tabelle auch mit einer $2 \times (b+1)$ Tabelle auskommen, wodurch der Speicherbedarf in $\mathcal{O}(b)$ liegt.

Aufgabe 2: Knapsack

Gegeben sei eine Menge von n Objekten. Jedes Objekt o_i besitzt einen ganzzahligen Wert v_i und ein ganzzahliges Gewicht $w_i \in \mathbb{N}$. Außerdem ist ein Maximalgewicht $W \in \mathbb{N}$ gegeben. Gesucht ist eine Teilmenge von Objekten U, sodass $\sum_{o_i \in U} w_i \leq W$, wobei $\sum_{o_i \in U} v_i$ maximiert wird. Lösen Sie dieses Problem mittels dynamischer Programmierung! Ihr Programm sollte eine Laufzeit von $\mathcal{O}(n \cdot W)$ haben. Es reicht, wenn Sie den Wert $\sum_{o_i \in U} v_i$ einer optimalen Lösung U bestimmen.

Lösung: Wir füllen eine Tabelle D, wobei der Eintrag $D(i,j)$ dem größtmöglichen Wert der ersten i Objekte enthält unter der Berücksichtigung des maximalen Gewichts j. Den Wert einer optimalen Lösung ist dann in $D(n,W)$ gespeichert.

Basisfall: Für $1 \leq j \leq W$

$$D(1,j) = \begin{cases} v_1 & \text{if } w_i \leq j \\ 0 & \text{else.} \end{cases}$$

Rekursionsfall:

$$D(i,j) = \max \begin{cases} v_i + D(i-1, j-w_i) & \text{if } w_i \leq j \\ D(i-1,j) \end{cases}$$

Wir haben n Zeilen und W spalten. Pro Eintrag benötigen wir $\mathcal{O}(1)$ Zeit. Somit benötigt ein dynamisches Programm $\mathcal{O}(n \cdot W)$ Zeit. Die tatsächlichen Objekte, die zum Wert der optimalen Lösung gehören, lassen sich mittels Backtracking bestimmen.

Aufgabe 3: Längste Wege in azyklischen Graphen

Wir haben bereits gesehen, dass das Problem Längster Wege in allgemeinen Graphen nicht die Eigenschaft optimaler Teilstrukturen aufweist. Wir beschränken uns daher nun auf azyklische gerichtete Graphen. Es wird ein solcher Graph $G = (V, E)$ mit $E \subseteq V^2$ sowie eine Kantengewichtsfunktion $w : E \mapsto \mathbb{R}^+$ gegeben. Außerdem werden Start- und Endknoten $s, t \in V$ übergeben. Gesucht ist ein längster Weg von s nach t.

(a) Wir wollen zunächst zeigen, dass dieses eingeschränkte Problem die optimale-Teilstruktur-Eigenschaft hat. Dazu wollen wir nun beweisen, dass der längste s-t-Weg für einen Knoten $v \in V$ aus einem längsten s-v-Weg sowie einem längsten v-t-Weg besteht.

Hinweis: Es könnte hilfreich sein, sich nochmal zu überlegen, wieso das Problem *kürzester* Wege aus optimalen Teilstrukturen besteht.

Lösung: Angenommen wir haben einen längsten s-t-Weg und einen Knoten v auf diesem Weg gegeben. Wir teilen diesen Weg nun in einen s-v-Weg und einen v-t-Weg.
Wir nehmen außerdem zunächst an, dass der so gefundene s-v-Weg nicht der längste ist. Auf diesem Weg liegt kein Knoten, der auf dem v-t-Weg liegt, da wir sonst einen Kreis gefunden hätten. Der Graph ist jedoch azyklisch. Also können wir den längeren s-v-Weg mit dem v-t-Weg kombinieren, um einen noch längeren s-t-Weg zu erhalten. Das widerspricht der Annahme.
Analog können wir auch argumentieren, dass der v-t-Weg bereits der längste ist.

(b) Was sind die Teilprobleme, die in einem dynamischen Programm gelöst werden müssten? Wie können diese Teilprobleme rekursiv gelöst werden?

Lösung: Es ist ausreichend, für jeden Knoten $v \in V$ einen längsten v-t-Pfad zu berechnen. Da der Graph azyklisch ist, ist der Längste t-t-Pfad dabei einfach der leere Pfad. Für $v \in V \setminus \{t\}$ können wir dann den Nachbarn u mit längstem u-t-Pfad wählen und die Kante (v, u) an den Anfang dieses Pfades einfügen.

(c) In welcher Reihenfolge müssen die Teilprobleme gelöst werden?

Lösung: Wir können den Graphen vor der Berechnung topologisch sortieren und in dieser Reihenfolge abarbeiten. Dadurch ist garantiert, dass bereits für alle Nachbarn des aktuell betrachteten Knotens ein längster Weg zu t berechnet ist.

Aufgabe 4: Independent Sets in Bäumen

Eine unabhängige Menge in einem Graphen ist eine Teilmenge $U \subseteq V$, sodass keine Knoten in U miteinander benachbart sind. Im Independent Set Problem ist nach der größten unabhängigen Menge in einem Graphen G gesucht. Im Allgemeinen ist dieses Problem NP-schwer, doch in Bäumen lässt sich dieses Problem in Polynomialzeit lösen. Sei also ein Baum $T = (V, E)$ gegeben. Schreiben Sie ein dynamisches Programm, das in $\mathcal{O}(V + E)$ Zeit die Kardinalität eines größten Independent Sets in T findet.

Hinweis 1: Ein Baum als Graph besitzt keine eindeutige Wurzel. Sie können annehmen, dass der Baum T bereits an Wurzel r gewurzelt ist.

Hinweis 2: Nutzen Sie die rekursive Struktur eines Baums aus! Nutzen Sie zwei Tabelleneinträge pro Knoten, einmal für den Fall, dass Sie den Knoten nehmen und einmal für den Fall, dass Sie ihn nicht nehmen.

Lösung: Sei $T = (V, E)$ ein Baum gewurzelt mit Wurzel $r \in V$. Sei $A(v)$ die Kardinalität eines größten Independent Sets im Teilbaum T_v mit Wurzel v und sei $B(v)$ die Kardinalität eines größten Independent Sets im Teilbaum T_v mit Wurzel v, wobei v nicht im Independent Set enthalten sein darf. Aus der Definition folgt, dass $A(r)$ das gewünschte Ergebnis liefert.

Basisfall: Der Basisfall tritt für alle Blätter in T ein. Nachdem der Teilbaum T_v eines Blatts v nur aus einem Knoten besteht gilt $A(v) = 1$ und $B(v) = 0$.

Rekursionsfall: Durch eine Postorder Berechnung können wir annehmen, dass $A(u)$ und $B(u)$ für alle Kinder von v bereits berechnet worden sind. Für den Fall, dass v nicht im Independent Set enthalten sein darf, müssen wir uns um nichts kümmern, da die Teilbäume der Kinder von v nicht miteinander verbunden sein können. Somit ist

$$B(v) = \sum_{u \text{ child of } v} A(u)$$

Falls wir v mit ins Independent Set wählen, darf kein Kind von v im Independent Set enthalten sein, denn andernfalls wären das Kind u und v benachbart. Falls wir v nicht wählen, können wir einfach $B(v)$ ausrechnen und diesen Wert auch für $A(v)$ benutzen:

$$A(v) = \max \left\{ B(v), 1 + \sum_{u \text{ child of } v} B(u) \right\}$$

Für jeden Knoten v summieren wir über Werte der Kinder. Somit haben wir insgesamt einen linearen Aufwand und erhalten eine Laufzeit von $\mathcal{O}(V + E)$.

Aufgabe 5: Der schönste Binärbaum der Welt

Sie sind auf Kiliani und laufen an einem Stand vorbei. Dort fällt Ihnen der schönste Binärbaum in die Augen, den Sie je gesehen haben. Unverzüglich gehen Sie den kürzesten Pfad zum Stand und sehen, dass dieser Binärbaum der Hauptpreis eines Spieles ist. Das Spiel funktioniert wie folgt:
Innerhalb des Standes sind n Pins nebeneinander aufgestellt. Auf jedem dieser Pins ist eine ganze Zahl (also auch negative) geschrieben. Sie erhalten einen Ball und müssen die Pins abwerfen, sodass Sie ihren Score maximieren. Dieser Score berechnet sich folgendermaßen:

- Sie treffen genau einen Pin: Sie erhalten den Wert des Pins aufaddiert zu ihrem bisherigen Score
- Sie treffen zwei benachbarte Pins: Sie erhalten das Produkt beider Pins zu ihrem bisherigen Score aufaddiert

Sie können nicht mehr als zwei benachbarte Pins treffen und Sie müssen nicht alle Pins abwerfen. Sobald Sie einen Pin abgeworfen haben, fällt dieser um und kann nicht erneut abgeworfen werden. Wie es der Zufall will können Sie ausgezeichnet werfen und treffen immer wohin Sie zielen.

Entwickeln Sie ein dynamisches Programm, das eine Sequenz $v_1, \ldots, v_n$ von Zahlen (die Werte der Pins) entgegen nimmt und den maximalen Score errechnet.

(a) Definieren Sie das Teilproblem $B(\cdot)$.

Lösung: $B(i) \hat{=}$ maximal möglicher Score für Pins $1, \ldots, i$

(b) Wie kann das Teilproblem, das Sie in Teilaufgabe (a) definiert haben berechnet werden? Geben Sie hierzu eine rekursive Gleichung an.

Lösung: Wir haben drei Möglichkeiten für Pin i: Wir werfen Pin i nicht ab, wir werfen lediglich Pin i ab oder wir werfen Pin i mit seinem Nachbarn ab. Nachdem Pin i in unserem Teilproblem der Pin am rechten Rand ist, hat Pin i nur einen linken Nachbarn. Nachdem $B(i)$ den maximalen Score angibt müssen wir von diesen drei Möglichkeiten das Maximum wählen. Demnach können wir $B(i)$ folgendermaßen rekursiv definieren:

$$B(i) = \begin{cases} \max\{B(i-1), B(i-1) + v_i, B(i-2) + v_{i-1} \cdot v_i\} & \text{falls } 2 \leq i \leq n \\ \max\{B(i-1), B(i-1) + v_i\} & \text{falls } i = 1 \\ 0 & \text{falls } i = 0 \end{cases}$$

Wir definieren den Basisfall für eine leere Sequenz mit einem maximal möglichen Score von 0.

(c) Schreiben Sie einen Pseudocode, der die rekursive Gleichung in Teilaufgabe (b) implementiert! Ordnen Sie die asymptotische Laufzeit des Algorithmus ein und begründen Sie ihre Antwort! (eine genaue Berechnung ist nicht gefordert)

Lösung: Im Prinzip müssen wir hier nur die Rekursionsgleichung in Pseudocode übertragen:

Algorithmus 2: getMaxBowlingScore(**int**[] v, **int** i)

```
1 if i == 0 then
2     return 0
3 if i ≥ 2 then
4         // Hier muss sichergestellt werden, dass wir kein Index Out Of Bounds
      bekommen
5     return max{getMaxBowlingScore(v, i − 1), getMaxBowlingScore(v, i − 1) + v[i],
      getMaxBowlingScore(v, i − 2) + v[i − 1]· v[i]}
6 else
7     return max{getMaxBowlingScore(v, i − 1), getMaxBowlingScore(v, i − 1) + v[i]}
```

Laufzeit: Die Laufzeit ist mindestens exponentiell, da die Rekursionsgleichung einen Teil $T(n) = 2T(n-1)+T(n-2)+\Theta(1)$ hat. Diese Gleichung hat eine große Ähnlichkeit zu den Fibonaccizahlen und diese wachsen bereits exponentiell. Genauer kann man die Rekursionsgleichung aber auch mit $T(n) \leq 3T(n-1)$ abschätzen, was einer Laufzeit von $\mathcal{O}(3^n)$ entspricht.

(d) Nutzen Sie nun *memoization*, um redundante Berechnungen in Ihrem rekursiven Algorithmus zu vermeiden! Welche Laufzeit hat der *memoized* Algorithmus nun?

Lösung: Wir lassen den Wrapper um die Funktion weg (Siehe Vorlesung).

Algorithmus 3: getMaxBowlingScore(**int**[] v, **int** i, **int**[] memo)

```
1  // memo ist ein Array der Länge n+1 und ist zu Beginn mit ∞ gefüllt
2  if memo ≠ ∞ then
3      return memo[i]
4  if i == 0 then
5      return 0
6  if i ≥ 2 then
7      memo[i] = max{getMaxBowlingScore(v, i − 1), getMaxBowlingScore(v, i − 1) + v[i],
       getMaxBowlingScore(v, i − 2) + v[i − 1]· v[i]}
8  else
9      memo[i] = max{getMaxBowlingScore(v, i − 1), getMaxBowlingScore(v, i − 1) + v[i]}
10 return memo[i]
```

Laufzeit: Durch die memo Tabelle sparen wir uns redundante Berechnungen ⇒ Laufzeit ist Anzahl Teilprobleme · Die Zeit die wir benötigen, ein Teilproblem zu lösen: $\Theta(n) \cdot \Theta(1) = \Theta(n)$

(e) Implementieren Sie den Algorithmus nun bottom-up!

Lösung: Beim bottom-up Ansatz füllen wir nun memo iterativ:

Algorithmus 4: getMaxBowlingScore(**int**[] v, **int** i)

```
1 int[] B = new int[n + 1]
2 B[0] = 0
3 B[1] = max{B[0], B[0] + v[1]}
4 for i = 2 to n do
5     B[i] = max{B[i − 1], B[i − 1] + v[i], B[i − 2] + v[i − 1]· v[i]}
6 return B[n]
```

Aufgabe 6: Perfekte Binärbäume

Angenommen wir haben n Elemente und wissen, mit welcher Wahrscheinlichkeit diese angefragt werden. Wir wollen nun einen binären Suchbaum finden, der die erwartete Anfragezeit minimiert. Das heißt, gegeben eine Wahrscheinlichkeitsverteilung der n Elemente $p_1, p_2, \ldots, p_n$ mit der die Elemente angefragt werden. Wir wollen die Suchzeit $\sum_{i=1}^{l} p_i \cdot l_i$ minimieren, wobei l_i das Level von Element i ist. Im Folgenden wollen wir ein dynamisches Programm entwickeln, das dieses Problem löst.

(a) Wie verändert sich die erwartete Suchzeit eines Teilbaums, wenn dieser an einen weiteren Knoten gehängt wird?

Lösung: Sei T_{ij} ein Teilbaum, der die Elemente $v_i, \ldots, v_j$ enthält, wobei dieser eine erwartete Suchzeit von $\sum_{k=i}^{j} p_k \cdot l_k$ besitzt. Wenn nun dieser Teilbaum an einen weiteren Knoten gehängt wird, erhöht sich das Level aller Knoten im Teilbaum T_{ij} um 1. Somit gilt die erwartete Suchzeit nun:

$$\sum_{k=i}^{j} p_k(l_k + 1) = \sum_{k=i}^{j} p_k \cdot l_k + \sum_{k=i}^{j} p_k.$$

(b) Angenommen Sie haben eine Methode, um die Wurzel eines optimalen Teilbaums zu berechnen. Gegeben sei ein Teilbaum mit folgender Sequenz von Elementen $v_i \ldots, v_r, \ldots, v_j$, wobei v_r die Wurzel eines

optimalen Teilbaums ist. Wie können Sie den Rest des Teilbaums berechnen?

Lösung: Seien die Elemente in aufsteigender Reihenfolge sortiert. Wenn wir nun die Wurzel v_r eines optimalen Teilbaums kennen, so können wir die Sequenz an der Wurzel spalten, sodass wir zwei kleinere Teilprobleme erhalten. Das heißt, wir können rekursiv den linken Teilbaum von v_r mit den Elementen $v_i, \ldots, v_{r-1}$ berechnen und wir können rekursiv den rechten Teilbaum von v_r mit den Elementen $v_{r+1}, \ldots, v_j$ berechnen und deren optimalen Wert mittels der Gleichung aus Teilaufgabe (a) anpassen.

(c) Sei $D[i, j]$ der Wert einer optimalen Lösung mit den Elementen $v_i, \ldots, v_j$. Stellen Sie die Rekursionsgleichung für $D[i, j]$ auf!

Lösung: Aus der vorherigen Teilaufgabe geht bereits hervor, welche Form die Rekursionsgleichung haben sollte. Allerdings kennen wir die optimale Wurzel v_r in einer Sequenz $v_i, \ldots, v_j$ bei der Berechnung noch nicht. Deshalb probieren wir alle möglichen Wurzeln aus und nehmen davon die jenige Wurzel, die die erwartete Suchzeit minimiert.

$$D[i,j] = \min_{i \le r \le j} \{ \underbrace{D[i, r-1] + \sum_{k=i}^{r-1} p_k}_{\text{linker Teilbaum mit (a)}} + p_r + \underbrace{D[r+1, j] + \sum_{k=r+1}^{j} p_k}_{\text{rechter Teilbaum mit (a)}} \}$$

$$= \min_{i \le r \le j} \{ D[i, r-1] + D[r+1, j] + \sum_{k=i}^{j} p_k \}$$

(d) Für welche Sequenz von Elementen kennen wir bereits die optimale erwartete Suchzeit (Basisfall)?

Lösung: Wenn wir eine leere Sequenz von Elementen haben, wissen wir sofort, dass $D[i, i-1] = 0$.

(e) Welche Laufzeit hat Ihr dynamisches Programm?

Lösung: Wir müssen nicht jedesmal $\sum_{k=i}^{j} p_k$ neu berechnen. Wir können die Summe vorberechnen, indem wir für jedes $1 \le i \le n$ die Summe $\sum_{k=1}^{i} p_k$ berechnen und den Wert in ein Feld an Stelle $S[i]$ speichern. Wenn wir nun $\sum_{k=i}^{j} p_k$ berechnen wollen, können wir einfach $S[j] - S[i-1]$ berechnen. Damit können wir einen rekursiven Aufruf in $\mathcal{O}(n)$ berechnen. Unsere Tabelle hat $\mathcal{O}(n^2)$ Einträge und somit haben wir eine Gesamtlaufzeit von $\mathcal{O}(n^3)$.

Aufgabe 7: Kürzeste Wege mit negativen Kanten

Gestern haben wir festgestellt, dass die Ergebnisse von Dijkstra auf Graphen mit negativen Kanten unter Umständen nicht die kürzesten Wege repräsentieren. In dieser Aufgabe wollen wir einen Algorithmus finden, der auf einem Graphen $G = (V, E)$ mit der Gewichtsfunktion $\mathrm{w} : V \times V \to \mathbb{R}$ einen kürzesten Weg zwischen zwei Knoten s und t findet. Wir nehmen an, dass der Graph G keine von s erreichbaren, negativen Kreise enthält.

(a) Zeigen Sie, dass das Problem eine optimale Substruktur aufweist. Sie müssen also zeigen, dass der kürzeste Weg zwischen s und t aus kleineren Teillösungen desselben Problems berechenbar ist. In welchen Fällen ist es besonders einfach, den kürzesten Weg zwischen s und t zu berechnen?

Lösung: Auf dem Weg von s nach t überqueren OdBA wir einen Knoten w. Also setzt sich der gesuchte kürzeste Weg von s nach t aus zwei Wegen zusammen, einem Weg von s nach w und

einem Weg von w nach t. Die beiden Teilwege sind jeweils kürzeste Wege und diese sind jeweils mindestens eine Kante kürzer.

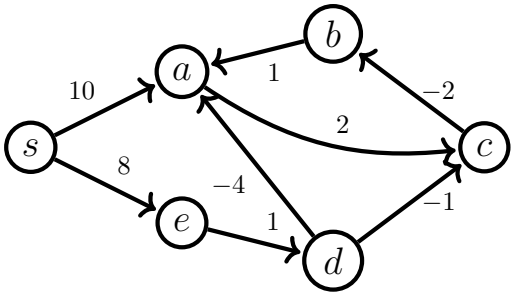
Damit erhalten wir ein kleineres Problem derselben Art, nämlich den kürzesten Weg von s zu einem Knoten w zu finden.

Besonders einfach ist das Problem zu lösen, falls $s = t$, da dann $\delta(s, t) = 0$.

(b) Wie viele Kanten kann jeder kürzeste Weg im Graphen $G = (V, E)$ maximal haben? Angenommen, der Distanzwert $v.d$ ist für alle $v \in V$ mit ∞ intialisiert und $s.d = 0$. Was passiert auf jeden Fall, wenn Sie die Relax-Methode (siehe Dijkstra) nun auf *jede* Kante in beliebiger Reihenfolge aufrufen? Was passiert, wenn Sie dies erneut tun?

Lösung: Der längste mögliche kürzeste Weg traversiert jeden Knoten maximal einmal. Deswegen ist die höchste mögliche Kantenanzahl $|V| - 1$. Nachdem Aufruf von Relax wurden die Distanzwerte der Nachbarn von s auf einen endlichen Wert gesetzt und mindestens einer von ihnen hat auch die korrekte Entfernung. Beim wiederholten Aufruf von Relax kommen immer mehr Knoten hinzu, deren Entfernung gesetzt ist und in jeder Iteration wird mindestens eine Entfernung richtig gesetzt.

(c) Wir betrachten nun eine zweidimensionale Tabelle T. Jede Spalte steht für einen Knoten (in beliebiger Reihenfolge), und es gibt $|V| - 1$ Zeilen. Die Zelle $T(i, j)$ enthält die *Länge* des kürzesten Weges von s zum i-ten Knoten, nachdem j Mal die Relax-Methode auf *alle* Kanten aufgerufen wurde. Stellen Sie die Tabelle für folgenden Graphen auf und füllen Sie sie zeilenweise aus. Relaxieren Sie die Kanten in alphabetischer Reihenfolge nach ihrem Startknoten.



Iteration	$s.d$	$a.d$	$b.d$	$c.d$	$d.d$	$e.d$
0	0	∞	∞	∞	∞	∞
1						
2						
3						
4						
5						

Lösung:

Iteration	$s.d$	$a.d$	$b.d$	$c.d$	$d.d$	$e.d$
0	0	∞	∞	∞	∞	∞
1	0	10	∞	∞	∞	8
2	0	10	10	12	9	8
3	0	5	10	11	9	8
4	0	5	5	7	9	8

(d) Geben Sie nun den Algorithmus in Pseudocode an. Welche Laufzeit hat er?

Lösung: Der Bellman-Ford-Algorithmus löst das Problem:

Algorithmus 5: BellmanFord(G, s, w)

```
1 for i = 1 to |V| − 1 do
2     foreach Kante (u, v) ∈ E do
3         if v.d > u.d + w(u, v) then
4             v.d = u.d + w(u, v)
```

Die Korrektheit kann durch die Schleifeninvariante „Nach der i. Iteration gilt für $i+1$ Knoten v, dass ihre berechnete Distanz $v.d$ mit dem tatsächlich kürzesten Weg $\delta(s, v)$ übereinstimmt." gezeigt werden.

(e) Modifizieren Sie Ihren Algorithmus so, dass er auch die optimale Lösung selbst, also den kürzesten Weg berechnet. Welche Methode aus der Vorlesung können Sie dafür verwenden?

Lösung: Wir ergänzen den Anweisungsblock der **if**-Abfrage um die Zeile $v.p = u$, wobei das Attribut $v.p$ jeweils auf den Elternknoten u auf dem bisher kürzesten gefundenen Wegs von s nach v zeigt. Die **if**-Abfrage entspricht nun genau der Relax-Funktion, die vom Dijkstra-Algorithmus bekannt ist.

(f) Unter welchen Umständen kann der Algorithmus vorzeitig abgebrochen werden? Wie kann mithilfe des Algorithmus ein negativer Zykel detektiert werden?

Lösung: Falls sich in einer Iteration keine d-Werte mehr ändern, kann der Algorithmus vorzeitig abgebrochen werden. Falls alle $|V| - 1$ Iterationen ausgeführt wurden und in einer letzten, $|V|$. Iteration noch d-Werte verändert werden, dann liegt ein negativer Zykel vor.

Aufgabe 8: Palindrome Subsequenzen

Ein *Palindrom* ist eine Zeichenkette, die von vorne und von hinten gelesen das gleiche ergibt, zum Beispiel das Adjektiv „soldlos". Eine *Subsequenz* ist ein String, der nach Weglassen beliebig vieler Zeichen aus einem String hervorgeht, beispielsweise das Wort „Baum" aus „Brauchtum". Wir suchen nun einen effizienten Algorithmus, der eine längste Subsequenz einer Zeichenkette $s = s_0 \dots s_n$ findet, die gleichzeitig ein Palindrom ist. Die längste palindrome Subsequenz in „Amortisierte Laufzeit" ist „tieteit".

(a) Erklären Sie kurz, wie ein Brute-Force-Algorithmus vorgehen würde, um das Problem zu lösen. Was ist die Laufzeit dieses Algorithmus?

Lösung: Der Brute-Force-Ansatz wäre, alle Subsequenzen aus s zu berechnen und diese auf ihre Palindrom-Eigenschaft zu testen. Da es jedoch 2^{n+1} Subsequenzen gibt, ist dieser Ansatz nicht effizient. Die Laufzeit wäre in $\mathcal{O}(n \cdot 2^{n+1})$.

(b) Gegeben sei eine Zeichenkette $s = s_0 \dots s_n$ und ihre längste palindrome Subsequenz $p = p_0 \dots p_m$. Beschreiben Sie wie p aus einer kleineren Instanz desselben Problems hervorgeht. Betrachten Sie dazu einen Teilstring s' von s, und erklären Sie, wie p zu s' steht.

Lösung:

1. Falls $s_0 \neq p_0$, dann ist p eine optimale Lösung für $s_1, \dots, s_n$.
2. Falls $s_n \neq p_m$, dann ist p eine optimale Lösung für $s_0 \dots s_{n-1}$.
3. Falls $s_0 = s_n$, dann ist $s_0 = p_m$ und $p_1 \dots p_{m-1}$ ist eine optimale Lösung in $s' = s_1 \dots s_{n-1}$.

Beweis. Es trifft immer genau einer der obigen Fälle zu. Wir begründen nun, dass die obigen Implikationen korrekt sind:

1. Analog zu ii.
2. Angenommen, p ist keine optimale Lösung für $s_0 \dots s_{n-1}$. Dann gibt es eine andere optimale Lösung mit der Länge größer als $m+1$ für $s_0 \dots s_{n-1}$. Das widerspricht aber der Annahme, dass p optimal für $s_0 \dots s_{n-1}$ ist.

3. Falls $s_0 \neq p_m$, dann könnten wir p verbessern, indem wir die offenbar noch nicht genutzten s_0 und s_n vorne und hinten an p anhängen. Das widerspricht aber der Annahme, das p schon optimal ist. Also muss $s_0 = p_m$ sein. Nun, nehmen wir an, $p_1 \dots p_{m-1}$ ist keine optimale Lösung in $s_1 \dots s_{n-1}$. Dann muss es eine andere optimale Lösung für $s_1 \dots s_{n-1}$ geben, die länger als $m-1$ ist. Da $s_0 = s_n$ könnten wir diese Lösung nutzen, um eine längste palindrome Subsequenz der Länge $m+2$ für s zu finden. Dies widerspricht ebenfalls der Annahme, das p optimal ist.

□

Die obigen Implikationen zeigen, dass wir das Problem der längsten palindromen Subsequenz immer auf kleine Substrings übertragen können. Die Teilprobleme überlappen sich, da wir zum Beispiel den inneren Teil von s in allen obigen Implikationen untersuchen müssen.

(c) Wir definieren $l(i, j)$ als die Länge der längsten palindromen Subsequenz im Substring $s_i \dots s_j$. Was ist $l(i, i)$ und $l(i, i+1)$? Dies sind die Basisfälle und sind einfach anzugeben. Überlegen Sie sich nun, wie Sie für allgemeine i, j mit $i < j$ den Wert $l(l, j)$ berechnen können. *Tipp*: Machen Sie eine Fallunterscheidung nach $s_i = s_j$ bzw. $s_i \neq s_j$ und greifen Sie auf $l(i', j')$ zu, wobei $i' < i$ oder $j' < j$.

Lösung: Lege eine Matrix der Größe $n+1 \times n+1$ an und fülle jede Diagonale, angefangen von der Hauptdiagonale, sukzessive nach oben rechts. Andere Richtungen sind denkbar, dann müssen die Indizes angepasst werden.

$$l(i,j) = \begin{cases} 1 & \text{falls } i = j \\ 2 & \text{falls } i+1 = j \text{ und } s_i = s_j \\ \max(l(i+1,j), l(i,j-1)) & \text{falls } s_i \neq s_j \\ l(i+1,j-1)+2 & \text{falls } s_i = s_j \end{cases}$$

(d) Legen Sie eine Matrix, die $l(i, j)$ für alle $0 \leq i \leq j \leq n$ repräsentiert, für die beiden Sequenzen „anna“ und „graphalgo“ an und füllen Sie sie aus. Wie lang sind die längsten palindromen Subsquenzen in den Wörtern? Wo steht der Wert der Lösung in der Matrix?

Lösung: Da „anna“ selbst ein Palindrom ist, ist die längste palindrome Subsequenz vier Zeichen lang. Die längste palindrome Subsequenz von „graphalgo“ ist 5.

	A	N	N	A
A	1	1	2	4
N		1	2	2
N			1	1
A				1

	G	R	A	P	H	A	L	G	O
G	1	1	1	1	1	3	3	5	5
R		1	1	1	1	3	3	3	3
A			1	1	1	3	3	3	3
P				1	1	1	1	1	1
H					1	1	1	1	1
A						1	1	1	1
L							1	1	1
G								1	1
O									1

Der Wert der optimalen Lösung steht bei einer komplett ausgefüllten Matrix in der ersten Zeile ganz rechts.

(e) Formulieren Sie jetzt einen Algorithmus, der eine solche Matrix automatisch ausfüllt und den *Wert* der Lösung zurückgibt.

Lösung: Die obige Matrix-Rekurrenz wird direkt umgesetzt. Die erste Fallunterscheidung entfällt hierbei durch die Initialisierung der Matrix in Zeile 3.

Algorithmus 6: findLongestPalindromeSubsequenceLength(String s)

```
1  n = s.length
2  values = new int [s.length][s.length]
3  Fülle Hauptdiagonale von values mit 1ern
4  for row = 1 to n - 1 do
5      currRow = 1
6      for col = row + 1 to n do
7          if s_currRow == s_col ∧ currRow + 1 ≠ col then
8              values[currRow][col] = values[currRow+1][col-1] + 2
9          else if s_currentRow == s_col then
10             values[currRow][col] = 2
11         else
12             values[currRow][col] = max (values[currRow][col-1], values[currRow+1][col])
13         currRow = currRow + 1
14 return values[1][n]
```

(f) Jetzt möchten Sie nicht nur den Wert ermitteln, sondern auch die längste palindrome Subsequenz selbst. Beschreiben Sie, wie Sie mit einer zweiten Matrix die längste palindrome Subsequenz ermitteln können.

Lösung: Wir legen eine Matrix an, die genau so groß ist wie die erste und füllen sie parallel mit der Hauptmatrix mit Pfeilen aus, die auf das Feld zeigen, auf dem der Wert des aktuellen Feldes basiert. Dann verfolgen wir den Weg vom obersten rechten Feld zurück, bis wir auf ein Feld treffen, das den Wert 1 oder zwei 2 hat. Diese beiden Felder haben laut Matrix-Rekkurenz keine Vorgänger-Felder. Spalten-Indizes, die von Feldern abhängen, die diagonal von einem anderen Feld abhängen, beschreiben Zeichen, die in der längsten palindromen Subsequenz vorkommen. Das heißt, wir speichern diese Felder. Sobald wir das Ende des Weges erreicht haben, fügen wir noch das letzte Zeichen hinzu. Nun haben wir die Hälfte des gesuchten Palindroms und müssen dies nur noch spiegeln. Dabei müssen wir auf gerade und ungerade Länge aufpassen (siehe Pseudocode).

(g) Zeichnen Sie auch die ergänzte Matrix für die Wörter „anna“ und „graphalgo“. Was ist die jeweils längste palindrome Subsequenz?

Lösung: Für „anna“ ist die Lösung „anna“, für „graphalgo“ ist die Lösung „gahag“.

	A	N	N	A
A	⊙	⊙	↓	↙
N		⊙	⊙	←
N			⊙	⊙
A				⊙

	G	R	A	P	H	A	L	G	O
G	⊙	←	←	←	←	↓	←	↙	←
R		⊙	←	←	←	↓	←	←	←
A			⊙	←	←	↙	←	←	←
P				⊙	←	←	←	←	←
H					⊙	←	←	←	←
A						⊙	←	←	←
L							⊙	←	←
G								⊙	←
O									⊙

(h) Ändern Sie Ihren Algorithmus so, dass er die längste palindrome Subsequenz zurückgibt.

Lösung:

Algorithmus 7: findLongestPalindromeSubsequence(String s)

```
1  n = s.length
2  values = new int [s.length][s.length]
3  directions = new int [s.length][s.length]
4  Fülle Hauptdiagonale von values mit 1ern
5  for row = 1 to n - 1 do
6      currRow = 1
7      for col = row + 1 to n do
8          if s_currRow == s_col ∧ currRow + 1 ≠ col then
9              values[currRow][col] = values[currRow+1][col−1] + 2
10             directions[currRow][col] = ↙
11         else if s_currentRow == s_col then
12             values[currRow][col] = 2
13             directions[currRow][col] = ←
14         else
15             if values[currRow][col-1] ≥ values[currRow+1][col] then
16                 values[currRow][col] = values[currRow][col−1]
17                 directions[currRow][col] = ←
18             else
19                 values[currRow][col] = values[currRow+1][col]
20                 directions[currRow][col] = ↓
21         currRow = currRow + 1
22 row = 1
23 length =  values[1][n]
24 col = length
25 result = „“
26 while values[row][col] ≠ 1 do
27     if directions[row][col] = ← then col = col − 1
28     if directions[row][col] = ↓ then row = row + 1
29     if directions[row][col] = ↙ then
30         row = row + 1
31         col = col − 1
32         result = result + s_col
33 if length mod2 == 0 then result = result + s_col + s_col + reverse(result)
34 else result = result + s_col + reverse(result)
35 return result
```

(i) Überlegen Sie sich, wie Sie Ihren Algorithmus ändern können, sodass er den längsten palindromen *Substring* findet. Im Wort „stirnlappenbasilisk“ ist der längste palindrome Substring „silis“, während die bisher betrachtete palindrome Subsequenz „silappalis“ ist. Formulieren Sie die Matrix-Rekurrenzen aus Teilaufgabe c) um und beschreiben Sie in Worten, wo sich in der Matrix jetzt der Wert der Lösung befindet und wie Sie die Lösung rekonstruieren können.

Lösung: Der Wert der Zelle $l(i, j)$ repräsentiert nun nicht mehr die optimale Lösung für den Substring $s_i \dots s_j$, sondern den Wert ein 1, falls $s_i \dots s_j$ kein Palindrom ist, andernfalls die Länge des Palindroms. Dementsprechend müssen wir lediglich verhindern, dass Werte nach oben durchgereicht

werden. Wir ändern also den dritten Fall entsprechend:

$$l(i,j) = \begin{cases} 1 & \text{falls } i = j \\ 2 & \text{falls } i+1 = j \text{ und } s_i = s_j \\ 1 & \text{falls } s_i \neq s_j \\ l(i+1, j-1) + 2 & \text{falls } s_i = s_j \end{cases}$$

Dann suchen wir nach dem höchsten Feld in der Matrix, welches dann auch gleich den Wert der optimalen Lösung entspricht. Nun gehen wir wieder diagonal nach links unten, bis wir auf ein leeres Feld treffen. Die Spalten-Indizes dieses Wegs sind die Buchstaben des Palindromes, die dann noch gespiegelt angehängt werden müssen.

(j) Falls Sie noch Zeit haben, geben Sie einen Brute-Force-Algorithmus an, der eine längste palindrome Subsequenz findet.

Lösung: Die Idee besteht darin, einen Branching-Algorithmus zu konsturieren.

Algorithmus 8: bruteForce($s_0 \dots s_n$, $p_0 \dots p_m = \varepsilon$)

```
1 if s_0 ... s_n = ε then
2     return True falls p_0 ... p_m Palindrom, False sonst
3 return bruteForce(s_1 ... s_n, p_0 ... p_m s_0) ∨ bruteForce(s_1 ... s_n, p_0 ... p_m)
```