Aufgabensammlung ADS-Repetitorium 2021

Amortisierte Analyse – Dynamische Programmierung

Aufgabe 1: Bank-Schließfächer und Heaps

Bei einer Bank können Sie Schließfächer mieten. In jedes Schließfach passt genau ein Gegenstand. Der Service ist aber nicht kostenlos. Für das Einlagern des n. Gegenstands stellt die Bank $\log n$ Euro in Rechnung. Ebenfalls werden bei der Rückgabe des n. Gegenstands $\log n$ Euro fällig.

(a) Geben Sie die Gesamtkosten für das Ein- und Auslagern den n. Gegenstands in Θ-Notation an.

Lösung: $\Theta(\log n)$.

(b) Wie muss die Bank ihre Bezahlpolitik ändern, sodass das Abholen der Gegenstände kostenlos ist, die Bank aber trotzdem denselben Gewinn macht?

Lösung: Die Bank verlangt die Kosten für das Abholen des Gegenstands bereits bei der Einlagerung. Die Kosten für das Einlagern des n. Gegenstands sind dann $2 \log n$.

(c) Geben Sie die Kosten für das Ein- und Auslagern mit der neuen Bezahlpolitik in Θ-Notation an.

Lösung: Die Kosten für das Hinbringen bleiben in $\Theta(\log n)$, während die Kosten für das Abholen in $\Theta(1)$ liegen.

(d) Übertragen Sie Ihre Überlegungen aus den vorherigen Teilaufgaben auf einen MaxHeap. Zeigen Sie mit der Buchhaltermethode, dass die Insert-Methode amortisiert eine Laufzeit von $\mathcal{O}(\log n)$ und ExtractMax eine konstante Laufzeit hat.

Lösung: Wir zahlen für eine Einfügeoperation der i. Zahl $\hat{e}_i = 2 \log i$ und für das Löschen der i. Zahl $\hat{l}_i = 1$. Für das Einfügen von n Zahlen und das anschließende Extrahieren zahlen wir also $\hat{c} = \sum_{i=1}^{n} 2 \log i + \sum_{i=1}^{n} 1$. Vergleichen wir das mit den tatsächlichen Worst-Case-Kosten $c = \sum_{i=1}^{n} \log i + \sum_{i=1}^{n} \log i$, so erkennen wir, dass $\hat{c} > c$. Da $\hat{c} \in \mathcal{O}(n \log n)$ und auch die tatächlichen Kosten $c \in \mathcal{O}(n \log n)$, schließen wir daraus, dass Insert eine amortisierte Laufzeit von $\mathcal{O}(\log n)$ hat und ExtractMax amortisiert in $\mathcal{O}(1)$ liegt.

(e) Lösen Sie die Aufgabe nun mit der Potentialmethode.

Lösung: Die Potentialmethode bei n Elementen im Heap definieren wir mit $\Phi(n) = \sum_{i=1}^{n} \log i$, also der Summe aller Höhen im Heap. Damit ist $\Phi(0) = 0$ und es gibt keinen Wert n, sodass $\Phi(k) < 0$. Nun berechnen wir die amortisierten Kosten der Operationen:
Insert: $\hat{c}_i = c_i + \Phi(n) - \Phi(n-1) = \log n + \sum_{i=1}^{n-1} \log i + \log n - \sum_{i=1}^{n-1} \log i = 2 \log n \in \mathcal{O}(\log n)$
ExtractMax: $\hat{c}_i = c_i + \Phi(n-1) - \Phi(n) = \log n + \sum_{i=1}^{n-1} \log i - (\sum_{i=1}^{n-1} \log i + \log n) = 0 \in \mathcal{O}(1)$

Aufgabe 2: 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 $w : V \times V \to \mathbb{R}$ 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.

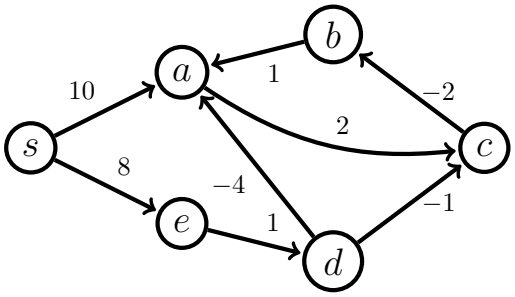
Damit erhalten wir ein kleineres Problem derselben Art, nämlich den kürzesten Weg von s zu einem Knoten w zu finden.

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.
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 3: 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:

i. Falls $s_0 \neq p_0$, dann ist p eine optimale Lösung für $s_1, \dots, s_n$.

ii. Falls $s_n \neq p_m$, dann ist p eine optimale Lösung für $s_0 \dots s_{n-1}$.

iii. 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:

i. Analog zu ii.

ii. 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.

iii. 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 1: 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 2: 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 3: 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)
```