Algorithmen 1 **1. Klausur** **Sommersemester 2023**

31.08.2023

Bitte klebe hier den Aufkleber mit deiner Matrikelnummer auf.

- Bringe den Aufkleber mit deiner Matrikelnummer oben auf diesem Deckblatt an und beschrifte jedes Aufgabenblatt mit deiner Matrikelnummer.
- Diese Klausur umfasst 18 Seiten (diese Titelseite eingeschlossen) und 6 Aufgaben. Es können maximal 60 Punkte erreicht werden.
- Schreibe die Lösungen auf die Aufgabenblätter und Rückseiten. Am Ende der Klausur sind zusätzliche Leerseiten. Fordere zusätzliches Papier bitte nur an, falls du den gesamten Platz aufgebraucht hast.
- Es werden nur Lösungen gewertet, die mit dokumentenechten Stiften geschrieben sind.
- Als Hilfsmittel ist ein A4-Papier mit beliebigem Inhalt erlaubt.
- Die Tackernadel darf nicht gelöst werden.
- Die Bearbeitungszeit beträgt 2 Stunden.
- Schreibe nicht in die Tabelle auf dieser Seite.

Aufgabe	Mögliche Punkte	Erreichte Punkte
1	10	
2	10	
3	10	
4	10	
5	10	
6	10	
Gesamt	60	

1. **Kleinaufgaben** [10 Punkte]

Hinweis: $\log(n) = \log_2(n)$ und $\log^2(n) = \log(n) \cdot \log(n)$.

(a) Ordne die folgenden Funktionen so an, dass genau dann $f \in O(g)$ gilt, wenn f links von g steht.(3 Punkte)

$$n \cdot \sqrt{n}, \qquad 2^n, \qquad 3 \cdot \log^4(n), \qquad n^2 + \sqrt{n}, \qquad n \cdot \log(n), \qquad 4 \cdot n^3, \qquad n!$$

Lösung:

$$3 \cdot \log^4(n), \qquad n \cdot \log(n), \qquad n \cdot \sqrt{n}, \qquad n^2 + \sqrt{n}, \qquad 4 \cdot n^3, \qquad 2^n, \qquad n!$$

(b) Die Funktion FOO(n) ist durch den unten gegebenen Pseudocode definiert. Bestimme eine Funktion f in Abhängigkeit von n, sodass FOO(n) Laufzeit $\Theta(f(n))$ benötigt. Vereinfache f so weit wie möglich. Gehe davon aus, dass DOSOMETHING() $\Theta(1)$ Zeit benötigt. (1 Punkt)

```
1: function FOO(n : ℕ)
2:     for i from 1 to n do
3:         j := 1
4:         while j · j < n do
5:             DOSOMETHING()
6:             j := j + 1
```

Laufzeit:

Lösung:

$$f(n) = n \cdot \sqrt{n} = n^{3/2}$$

(c) Gib für die folgenden Fragestellungen jeweils die Laufzeit eines möglichst effizienten Algorithmus, der das Problem löst, in O-Notation an. (3 Punkte)

0. Beispiel: Sortiere n vergleichbare Objekte. $\rightarrow O(n \log(n))$
1. Bestimme die zweitkleinste Zahl in einem unsortierten Array mit n Zahlen.

   Lösung: $O(n)$

2. In einem Text der Länge n, bei dem an jeder Stelle einer von 26 möglichen Buchstaben steht, soll der Buchstabe bestimmt werden, der am häufigsten vorkommt.

   Lösung: $O(n)$

3. Addiere zwei quadratische Matrizen mit n Zeilen.

   Lösung: $O(n^2)$

(d) Bestimme für folgende Situationen, welche möglichst einfache und effiziente Datenstruktur jeweils geeignet ist und gib an, welche Operationen der von dir gewählten Datenstruktur dabei relevant sind. (3 Punkte)
Hinweis: In der Vorlesung haben wir folgende Datenstrukturen kennengelernt: Listen, (dynamische) Arrays, Hashtabellen, binäre Heaps, $(2, 3)$-Bäume, Union-Find, Adjazenzliste.

1. Eine Magierin hat viele Beutel mit Murmeln. Für jede Murmel weiß sie, in welchem Beutel sie sich befindet. Ein Zuschauer darf immer wieder einen Beutel aussuchen und alle Murmeln aus diesem Beutel in einen anderen Beutel seiner Wahl umfüllen. Die Magierin soll danach für jede Murmel sagen

können, in welchem Beutel sie sich befindet.

Datenstruktur:

Lösung: Union-Find

Operationen:

Lösung: Union und Find.

2. Der Mensakoch bekommt jeden Tag Paprikas geliefert, die verschiedene Ablaufdaten haben. Beim Kochen möchte er immer eine Paprika verwenden, die als nächstes abläuft.

   Datenstruktur:

   Lösung: Heap

   Operationen:

   Lösung: Einfügen, kleinstes Element löschen.

3. Alan arbeitet in einer großen Bibliothek mit vielen Büchern. Manchmal werden alte Bücher aussortiert und neue Bücher eingekauft. Er fragt sich häufig, wie viele Bücher in der Bibliothek von einer gegebenen Person verfasst wurden.

   Datenstruktur:

   Lösung: Hashmap (hier wäre $(2, 3)$-Baum auch okay)

   Operationen:

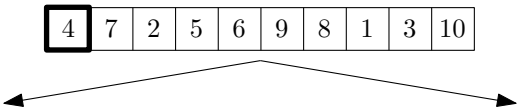
   Lösung: Einfügen, löschen, suchen.

2. **Sortieren** [10 Punkte]

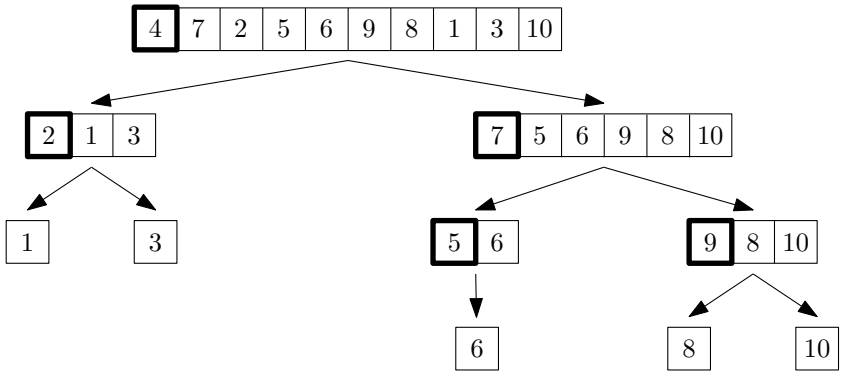
(a) Das folgende Array soll mit dem Quicksort-Algorithmus aus der Vorlesung sortiert werden, wobei als Pivotelement immer das *erste* Element eines Arrays gewählt wird.

4	7	2	5	6	9	8	1	3	10

Vervollständige den zugehörigen Rekursionsbaum, wobei jeder Knoten des Rekursionsbaums einem rekursiven Aufruf entspricht. Gib für jeden Knoten das zu sortierende Teilarray und das gewählte Pivotelement an. (3 Punkte)



Lösung:



(b) Sortiere folgendes Array mit dem LSD-Radix-Sort-Algorithmus mit Basis 10 aus der Vorlesung. Gib dabei für jede Stelle i der Dezimaldarstellung den Zustand der Buckets an. (4 Punkte)

521	423	125	755	645	741	789	332	123	842

Hinweis: Falls mehrere Zahlen in das gleiche Bucket gehören, schreibe sie untereinander in das richtige Bucket.

	0	1	2	3	4	5	6	7	8	9
$i = 0$										
$i = 1$										
$i = 2$										

Lösung:

	0	1	2	3	4	5	6	7	8	9
$i = 0$		521 741	332 842	423 123		125 755 645				789
$i = 1$			521 423 123 125	332	741 842 645	755			789	
$i = 2$		123 125		332	423	521	645	741 755 789	842	

(c) Wir betrachten nun Mergesort und die Methode `merge`(A_1, A_2). Wir nennen einen Merge von zwei Arrays A_1 und A_2 *verzahnt*, wenn niemals zwei aufeinander folgende Zahlen aus dem gleichen Array genommen

werden. Beispielsweise ist der Merge von $[1, 4, 8]$ mit $[2, 6, 9]$ verzahnt, der Merge von $[1, 4, 8]$ mit $[2, 3, 9]$ aber nicht. Gib ein Array A an mit den folgenden Eigenschaften:

- Es enthält die Zahlen $1, 2, 3, 4, 5, 6, 7, 8$ jeweils genau einmal.
- Bei der Ausführung von Mergesort auf dem Array A ist jeder Merge verzahnt.

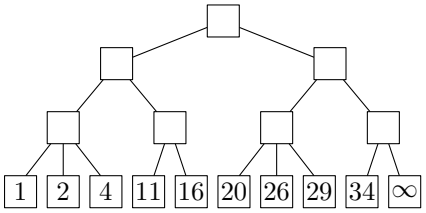
(3 Punkte)

Lösung: Eine mögliche Lösung ist $[1, 5, 3, 7, 2, 6, 4, 8]$.
Im Allgemeinen ist A genau dann eine Lösung, wenn folgendes gilt:

- Die Zahlenmengen $\{1, 5\}, \{3, 7\}, \{2, 6\}, \{4, 8\}$ sind jeweils in einem konsekutivem Teilarray von A enthalten und
- die Zahlenmengen $\{1, 3, 5, 7\}, \{2, 4, 6, 8\}$ sind jeweils in einem konsekutivem Teilarray von A enthalten.

3. **(2, 3)-Bäume** [10 Punkte]

In dieser Aufgabe beschäftigen wir uns mit $(2, 3)$-Bäumen, bei denen der ∞-Trick angewendet wird. Gegeben sei folgender $(2, 3)$-Baum:

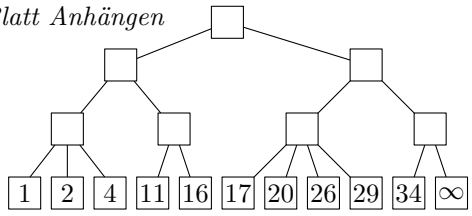


(a) Es sollen nun nacheinander die Operationen `insert(17)` und `remove(34)` ausgeführt werden. Gib die Subroutinen an, die jeweils durchgeführt werden müssen, um diese Operationen umzusetzen und anschließend die $(2, 3)$-Baum-Eigenschaft wiederherzustellen. Zeichne den Baum nach jeder Anwendung einer Subroutine.
Hinweis: Mögliche Subroutinen sind *Aufspalten*, *Verschmelzen*, *Blatt Löschen*, *Blatt Anhängen*, *Ausbalancieren*. Du brauchst keine Schlüssel in den inneren Knoten einzutragen. (4 Punkte)

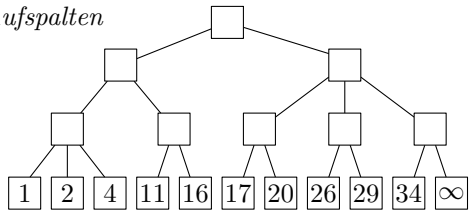
`insert(17)`:

Lösung:

Blatt Anhängen



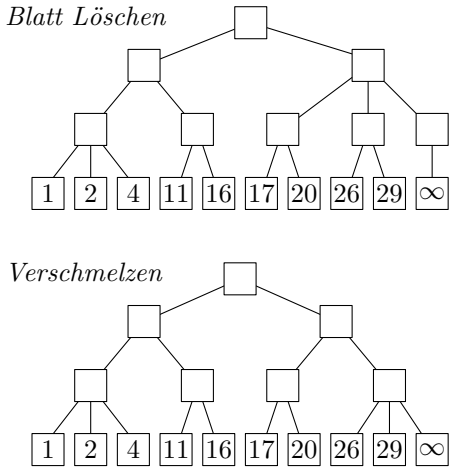
Aufspalten



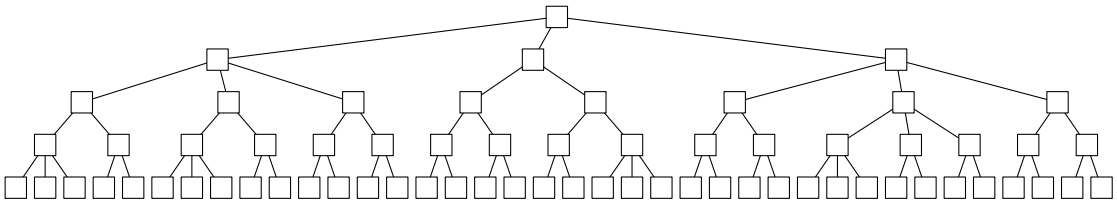
Zeichne den Zustand nach `remove(34)` oben auf der nächsten Seite.

`remove(34)` (nach `insert(17)`):

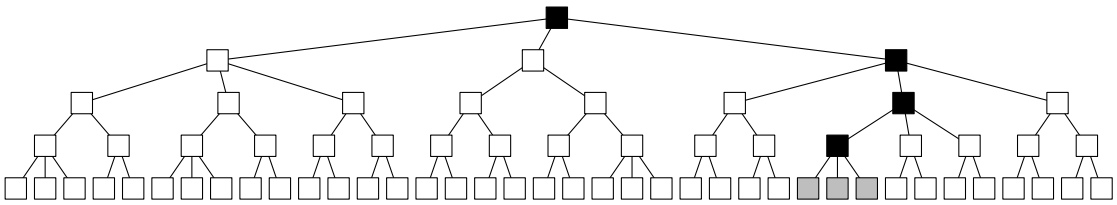
Lösung:



(b) Wir betrachten nun einen großen $(2, 3)$-Baum. Markiere alle Blätter v im Baum, sodass ein Einfügen vor v alle Knoten auf dem Pfad zur Wurzel aufspaltet. Markiere außerdem die Knoten, die dabei aufgespaltet werden. Gib zusätzlich an, unter welchen Bedingungen ein Einfügen vor einem Blatt w eines beliebigen $(2, 3)$-Baums alle Knoten auf dem Pfad zur Wurzel aufspaltet. (3 Punkte)



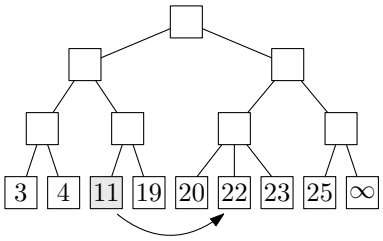
Lösung:



Bedingungen:

Lösung: Einfügen vor einem Blatt w spaltet genau dann alle Knoten auf dem Pfad zur Wurzel auf, wenn alle Knoten auf diesem Pfad (außer w) drei Kinder haben.

(c) Nun sollen $(2,3)$-Bäume etwas ausgedünnt werden. Wir sagen, dass ein Element x im Baum *verdoppelt vorkommt*, wenn es im Baum ein weiteres Element $y = 2 \cdot x$ gibt. Im folgenden Baum kommt Element 11 verdoppelt vor.



Beschreibe einen Algorithmus in Worten, der als Eingabe einen $(2,3)$-Baum erhält und unter allen verdoppelt vorkommenden Elementen das *kleinste* löscht. Nenne und begründe außerdem das asymptotische Laufzeitverhalten des Algorithmus. (3 Punkte)

Lösung:

Ansatz 1

Algorithmus: Wir iterieren über die sortierte Liste der Blätter im Baum in aufsteigender Reihenfolge. Für jedes Element x prüfen wir mittels `find(2x)`, ob es verdoppelt vorkommt. Falls ja, wird x gelöscht und der Algorithmus terminiert. Andernfalls passiert nichts und es wird mit dem nächsten Element fortgefahren.

Laufzeit: Das Iterieren der Liste benötigt $O(n)$ Zeit. In jeder Iteration wird ein Element gesucht und ggf. gelöscht, was beides in Zeit $O(\log(n))$ stattfindet. Somit ergibt sich insgesamt eine Laufzeit von $O(n \log(n))$.

Ansatz 2 (effizienter, aber weniger $(2,3)$-Baum bezogen)

Algorithmus: Im Verlauf des Algorithmus betrachten wir immer zwei ausgewählte Elemente e_s (klein) und e_ℓ (groß). Initial ist e_s das kleinste Element (ganz links in der Liste) und e_ℓ das zweit-kleinste.

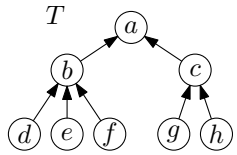
Die beiden Elemente laufen nun aufsteigend durch die Liste, bis $e_\ell = 2 \cdot e_s$ gilt oder das Ende der Liste erreicht ist. Nach jedem Schritt wird gepëuft, ob die Gleichung erfüllt ist. Wenn nicht, so ist eines der beiden Elemente zu klein und läuft daher zum nächsten Element in der Liste. Sobald die Gleichung erfüllt ist, wird e_s gelöscht und der Algorithmus terminiert.

Laufzeit: Jedes der beiden Elemente macht höchstens n Schritte. Dabei kann in konstanter Zeit überprüft werden, ob die Gleichung erfüllt ist und, wenn nicht, welches Element weiter läuft. Somit ergibt sich eine Laufzeit von $O(n)$ zum Finden eines verdoppelt vorkommenden Elements. Dazu kommt $O(\log(n))$ Zeit zum einmaligen Löschen. Insgesamt ist die Laufzeit also in $O(n)$.

Korrektheit: Die Korrektheit ergibt sich aus der Invariante, dass für ein gegebenes e_s alle potentiellen Verdopplungen rechts von e_ℓ liegen und, analog, alle potentiellen Halbierungen von e_ℓ rechts von e_s liegen.

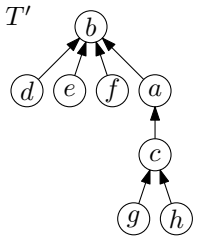
4. **Umwurzelung** [10 Punkte]

Wir sagen, ein Baum T ist an Knoten $r \in V(T)$ *gewurzelt*, wenn alle Kanten zu r hin zeigen (das heißt, es gibt von jedem Knoten einen gerichteten Pfad zur Wurzel). Für jeden Knoten $v \in V$ sei $|T_v|$ die *Teilbaumgröße von* v, also die Anzahl Knoten, von denen es einen Pfad zu v gibt (inklusive v). Im folgenden Beispiel ist für jeden Knoten v der Wert von $|T_v|$ angegeben.



Knoten	a	b	c	d	e	f	g	h
Teilbaumgröße	8	4	3	1	1	1	1	1

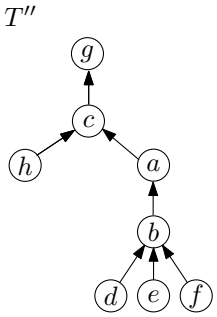
Eine *Umwurzelung* von der aktuellen Wurzel r zu einem Knoten v macht v zur neuen Wurzel und richtet die Kanten entsprechend. Wenn wir beispielsweise den oben gezeichneten Baum T von a zu b umwurzeln, erhalten wir folgenden gewurzelten Baum T'.



(a) Sei T'' der Baum, den man erhält, wenn man T stattdessen von a zu g umwurzelt. Gib die Teilbaumgrößen von T'' an. (1 Punkt)

Knoten	a	b	c	d	e	f	g	h
Teilbaumgröße								

Lösung:



Knoten	a	b	c	d	e	f	g	h
Teilbaumgröße	5	4	7	1	1	1	8	1

(b) Sei T ein Baum mit Wurzel r, sei c ein Kind der Wurzel und sei T' der Baum, den wir erhalten, wenn wir T von r zu c umwurzeln. Angenommen, du kennst für jeden Knoten v die Teilbaumgröße $|T_v|$ in T. Gib für jeden Knoten v in T' die Teilbaumgröße $|T'_v|$ in Abhängigkeit von den Teilbaumgrößen in T an.(3 Punkte)

Lösung:

$$|T'_c| = |T_r|$$
$$|T'_r| = |T_r| - |T_c|$$

sowie

$$|T'_v| = |T_v| \text{ für alle } v \in V \setminus \{c, r\}.$$

(c) Zeige die folgende Aussage: Für jedes $n > 2$ gibt es einen Baum mit n Knoten sowie zwei Knoten r_1 und r_2, sodass die Umwurzelung von r_1 zu r_2 die Teilbaumgröße eines Knotens um $\Theta(n)$ ändert. (2 Punkte)

Lösung: Betrachte einen Stern mit n Knoten, der zuvor am Zentrum gewurzelt war und wurzle ihn an einem Blatt. Jedes Blatt hat Teilbaumgröße 1. Die neue Wurzel hat dann Teilbaumgröße n im umgewurzelten Baum, was eine Änderung um $\Theta(n)$ ist.

Wir betrachten nun eine Datenstruktur, die einen Baum T mit Wurzel r verwaltet, dessen Knoten von 0 bis $n-1$ durchnummeriert sind. Diese Datenstruktur besteht aus

- einem Array `parent`, das den Baum repräsentiert, indem es an Stelle v den Elter von v speichert. Für die Wurzel r gilt: `parent[`r`]`$= r$.
- einem Array `tbg`, das an Stelle v die Teilbaumgröße von v speichert.

Wir gehen davon aus, dass diese Arrays zu Beginn korrekt initialisiert werden. Die Datenstruktur verfügt außerdem über eine Operation `swapRootToChild`(c), die den Baum in Zeit $\Theta(1)$ von der aktuellen Wurzel zum Kind c der Wurzel umwurzelt, indem sie das `parent`-Array und das `tbg`-Array entsprechend anpasst.

(d) Es soll nun eine neue Operation `swapRootToNode`(v) implementiert werden, die einen *beliebigen* Knoten v aus T entgegen nimmt und T von r zu v umwurzelt. Gib einen Algorithmus in Pseudocode an, der `swapRootToNode` umsetzt. Verwende folgende Signatur: (4 Punkte)

Hinweis: Fällt dir ein Algorithmus ein, dessen asymptotische Laufzeit nicht größer ist als die Distanz von v zu r?

`swapRootToNode`(v: $\mathbb{N}$) :

Lösung:

```
path: List
while parent[v] ≠ v do
    path.pushfront(v)
    v = parent[v]
for v ∈ path do
    swapRootToChild(v)
```

5. **Fête de la musique** [10 Punkte]

In einer Straße, in der nur Bands wohnen, wird ein Fest stattfinden. Aus den unterschiedlich beliebten Bands soll dazu eine gute Auswahl getroffen werden, um sie vor ihrer jeweiligen Haustür spielen zu lassen. Damit sich die Bands nicht gegenseitig übertönen, muss dabei aber darauf geachtet werden, dass zwei ausgewählte Bands nicht zu nah beieinander wohnen.

Unsere Eingabe besteht also aus den Bands $B = \{1, \ldots, n\}$, wobei Band i die Beliebtheit $p_i \in \mathbb{N}$ hat. Zudem ist ein Mindestabstand $d \in \mathbb{N}$ gegeben. Die *Beliebtheit* einer Teilmenge $S \subseteq B$ von Bands definieren wir als die Summe der Beliebtheiten der einzelnen Bands $p(S) = \sum_{i \in S} p_i$. Wir nennen S *gültig*, falls für alle Paare ausgewählter Bands $i, j \in S$ mit $i \neq j$ der Mindestabstand eingehalten wird, also wenn $|i - j| \geq d$.

Beim Problem d-BeliebtesMusikFest geht es darum, eine gültige Teilmenge mit möglichst großer Beliebtheit zu finden, wobei d der Mindestabstand ist. Eine solche Teilmenge nennen wir *optimal*.

Als Beispiel betrachten wir die folgende Instanz von 2-BeliebtesMusikFest zusammen mit einer gültigen, aber *nicht-optimalen* Teilmenge $S_1 = \{1, 3, 5, 7, 9\}$ mit Beliebtheit $p(S_1) = 2 + 6 + 1 + 8 + 2 = 19$.

Band i	1	2	3	4	5	6	7	8	9	
Beliebtheit p_i	2	1	6	8	1	1	8	1	2	
Enthalten in S_1	⊗	○	⊗	○	⊗	○	⊗	○	⊗	⟶ $p(S_1) = 19$

(a) Zeige, dass die Teilmenge S_1 im obigen Beispiel nicht optimal ist, indem du eine optimale Teilmenge S_2 angibst (bspw. durch Markieren in der unteren Tabelle).
Gib außerdem die Beliebtheit von S_2 an. (2 Punkte)

Band i	1	2	3	4	5	6	7	8	9	
Beliebtheit p_i	2	1	6	8	1	1	8	1	2	
Enthalten in S_2	○	○	○	○	○	○	○	○	○	⟶ $p(S_2) =$

Lösung: $S_2 = \{1, 4, 7, 9\}$ mit Beliebtheit $p(S_2) = 2 + 8 + 8 + 2 = 20$.

Für beliebiges $d > 0$ wollen wir nun ein dynamisches Programm für d-BeliebtesMusikFest entwickeln, wobei wir zuerst nur die Beliebtheit einer optimalen Teilmenge ermitteln wollen. Das DP soll für jeden Index $i \in B = \{1, \ldots, n\}$ eine Teillösung $X[i]$ in einem Array X verwalten.

(b) Gib an, welche Bedeutung eine Teillösung $X[i]$ hat und stelle darauf aufbauend die Rekurrenz auf, mit deren Hilfe das Array X korrekt ausgefüllt werden kann. (5 Punkte)
Hinweis: Achte darauf, auch den Basisfall der Rekurrenz anzugeben. Für volle Punktzahl sollte die Berechnung des DPs in $O(n)$ Zeit möglich sein.

Lösung: $X[i]$ entspricht der Beliebtheit einer optimalen Teilmenge der Bands $\{1, \ldots, i\}$.

$$X[i] = \begin{cases} \max_{1 \leq k \leq i}\{p_k\}, & \text{falls } i \leq d \\ \max\{X[i-1], X[i-d] + p_i\}, & \text{sonst.} \end{cases}$$

(c) Wir wollen nun nicht nur die Beliebtheit einer optimalen Teilmenge ermitteln, sondern auch die Menge selbst. Gib an,

- welche Informationen dafür während der Berechnung des dynamischen Programms gespeichert werden,
- wie die Lösung dann rekonstruiert werden kann und
- was die asymptotische Laufzeit dieser Rekonstruktion ist. (3 Punkte)

Lösung:

Information: Für jedes $i \in B = \{1, \ldots, n\}$ speichern wir uns, ob p_i in der optimalen Teilmenge von $\{1, \ldots, i\}$ enthalten ist. Dies ergibt sich direkt aus der Rekurrenz.

Rekonstruktion: Wir können dann die Lösung rekonstruieren, indem wir schauen, ob p_n in der optimalen Teilmenge enthalten ist und davon abhängig die Rekonstruktion bei $X[n-1]$ (falls p_n nicht enthalten ist) oder $X[n-d]$ (falls p_n enthalten ist) fortführen. Dies können wir iterativ fortsetzen, bis wir die gesamte Lösung rekonstruiert haben.

Laufzeit: Die Rekonstruktion erfolgt in $O(n)$ Zeit, da wir pro Index nur konstant viel Berechnungsaufwand haben.

6. **Straßenausbesserung** [10 Punkte]

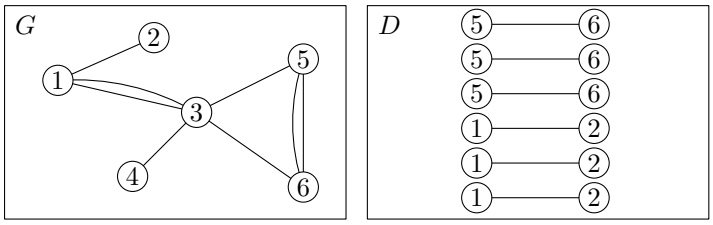
In einem Straßennetzwerk fallen häufig Reparaturen an, um Schlaglöcher auszubessern. Dazu wird auf einer Straße einfach eine komplett neue Schicht Asphalt aufgetragen. Sobald eine Straße jedoch aus mehr als k Schichten besteht (wobei $k \in \mathbb{N}$ mit $k > 0$), ist sie zu schwer. Um Platz für eine neue Schicht zu machen, werden die alten Schichten abgetragen und zur Schicht-Deponie gebracht. Nun sollen diese Ausbesserungsarbeiten mit einer Datenstruktur verwaltet werden.

Die Datenstruktur basiert auf einem *Multi*-Graphen G, der das Straßennetzwerk abbildet. Das heißt, in G kann es zwischen zwei Knoten *mehrere* Kanten geben. Jede Kante repräsentiert dabei eine Schicht. Zusätzlich wird die Deponie mithilfe eines Stacks D abgebildet. Die Datenstruktur unterstützt die folgenden Operationen:

- `dispose`(u, v): Falls es zwischen u und v in G mindestens eine Kante gibt, wird eine beliebige Kante zwischen u und v aus G entfernt und auf den Stack D gelegt, wofür $\Theta(1)$ Zeit benötigt wird.
- `coat`(u, v): Falls u und v in G bereits mit k Kanten verbunden sind, wird k-mal `dispose`(u, v) verwendet, um diese in Zeit $\Theta(k)$ auf den Stack D zu verschieben. In jedem Fall wird in Zeit $\Theta(1)$ zwischen u und v eine neue Kante in G eingefügt.
- `clear`(): Entfernt alle m_D Kanten, die aktuell auf dem Stack D liegen. Dafür wird $\Theta(m_D)$ Zeit benötigt.

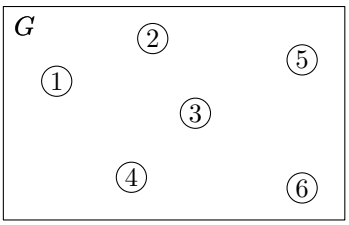
Im Folgenden gehen wir immer davon aus, dass G und D initial *keine* Kanten enthalten. Die Abbildung zeigt für $V = \{1, 2, 3, 4, 5\}$ und $k = 3$ den Zustand der Datenstruktur, den man erhält, wenn man die Operationen `coat`$(1, 2)$ 4-mal, `coat`$(1, 3)$ 2-mal, `coat`$(5, 6)$ 5-mal und `coat`$(3, 4)$, `coat`$(3, 5)$ und `coat`$(3, 6)$ jeweils 1-mal ausführt.

Zustand 1:



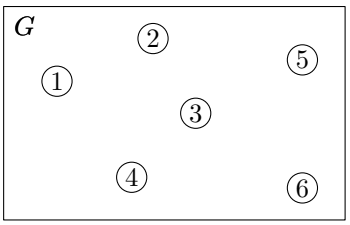
(a) Gib den Zustand der Datenstruktur an, den man erhält, wenn man ausgehend von *Zustand 1* die Operationen `coat`$(1, 3)$, `coat`$(1, 3)$, `dispose`$(1, 2)$, `coat`$(2, 5)$ und `coat`$(5, 6)$ ausführt. Zeichne dazu den Graphen G (oder gib die Kantenmenge an) und gib die Anzahl m_D von Kanten an, die in der Deponie D sind.(2 Punkte)

Zustand 2:



$m_D =$ ☐

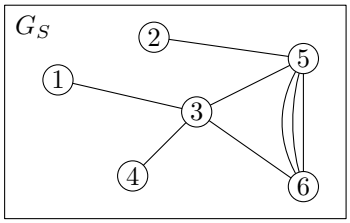
Zustand 2:



(Kopie desselben Graphen, falls du dich beim ersten vertan hast. Markiere deutlich, welche Lösung zu bewerten ist.)

Lösung:

Zustand 2:



$m_D =$ 10

(b) Im Folgenden gehen wir von einem Zustand aus, in dem die Datenstruktur keine Kanten enthält. Gib eine Knotenmenge V an und für jedes $n \in \mathbb{N}$ eine Sequenz von Operationen, sodass

- die Sequenz aus $\Theta(n)$ Operationen besteht,
- es eine `clear`-Operation gibt, die Laufzeit $\Theta(n)$ hat und
- die Sequenz $\Theta(n)$ viele `clear`-Operationen enthält.

Deine Konstruktion soll für jede Konstante k funktionieren. Begründe deine Antwort. (3 Punkte)

Hinweis: Achte darauf, dass n *nicht* die Anzahl der Knoten in V beschreibt.

Lösung: Wir wählen $V = \{u, v\}$ und führen folgende Sequenz von Operationen in drei Schritten aus:

1. n mal (`coat`(u, v), `dispose`(u, v), `clear`())
2. n mal (`coat`(u, v), `dispose`(u, v))
3. `clear`()

In den Schritten 1, 2 und 3 beträgt die Anzahl Operationen jeweils $3n$, $2n$ und 1, was in Summe $5n+1$ Operationen entspricht. Da $k > 0$ ist und auf jedes `coat` direkt ein `dispose` folgt, werden keine weiteren Operationen durch `coat` hervorgerufen. Somit besteht Sequenz aus genau $5n + 1 \in \Theta(n)$ Operationen.

In Schritt 1 werden genau $n \in \Theta(n)$ viele `clear`-Operationen ausgeführt.

Da in Schritt 2 erst n Kanten eingefügt und dann entfernt werden, befinden sich vor Schritt 3 genau $m_D = n$ Kanten in D. Die `clear`-Operation in Schritt 3 hat somit eine Laufzeit von $\Theta(m_D) = \Theta(n)$.

(c) Zeige, dass in jeder beliebigen Abfolge von `coat`-, `dispose`- und `clear`-Operationen jede Operation amortisiert konstante Kosten hat. (5 Punkte)

Lösung: Das kann mit Charging, Konto oder Potential gelöst werden.

Charging-Methode: Die Kosten m_D einer `clear` Operation chargen wir gleichmäßig auf die m_D `coat` Operationen, welche die Kanten eingefügt haben, die jetzt gelöscht werden. Damit ist `clear` amortisiert kostenlos.

Wird bei einer `coat` Operation k mal `dispose` aufgerufen (mit jeweils $\Theta(1)$ Laufzeit), dann chargen wir die Kosten für jeden Aufruf auf die vorherige `coat` Operation, welche die Kante eingefügt hat, die von `dispose` auf den Stack geschoben wird. Damit wird `coat` die Kosten für alle `dispose` Aufrufe los und `coat` kostet amortisiert $\Theta(1)$.

Da jede Kante nur einmal mit `dispose` auf den Stack verschoben und nur einmal mit `clear` gelöscht wird, werden so auf jedes `coat` nur zwei Kostentoken gecharged. Damit ist die amortisierte Laufzeit jeder Operation in $\Theta(1)$.

Konto-Methode: Jede `coat` Operation zahlt 2 auf das Konto ein. Jede `dispose` Operation hebt 1 ab und jede `clear` Operation hebt m_D ab. Damit erhalten wir die folgenden amortisierten Kosten. `dispose` und `clear` decken die tatsächlichen Kosten vollständig mit der Abhebung vom Konto und

sind damit amortisiert kostenlos. Falls eine `coat` Operation k mal `dispose` aufruft, dann fallen dafür amortisiert keine Kosten an, da `dispose` amortisiert kostenlos ist. Damit bleiben für `coat` $\Theta(1)$ tatsächliche Kosten plus $2 \in \Theta(1)$ für die Einzahlung.

Es bleibt zu zeigen, dass das Konto nie negativ wird. Sei dazu m_G die Anzahl der Kanten in G. Wir zeigen induktiv, dass der Kontostand immer mindestens $2m_G + m_D \geq 0$ beträgt. Das gilt offensichtlich zu Beginn, wenn $m_G = m_D = 0$. Wenn `clear` etwas abhebt, dann sinkt der Kontostand um m_D, aber außerdem sinkt auch die Anzahl Kanten in D um m_D auf 0. Damit bleibt die Invariante erhalten. Wenn `dispose` 1 abhebt, dann wird außerdem eine Kante von G nach D verschoben. Das heißt m_G sinkt um 1 und m_D wächst um 1. Wenn also vorher mindestens $2m_G + m_D$ auf dem Konto waren, dann gilt das auch hinterher. Bei `coat` erhöht sich m_G um 1, aber wir zahlen auch 2 aufs Konto ein, daher bleibt der Kontostand mindestens $2m_G + m_D$ (wir ignorieren hier die potentiellen `dispose` Aufrufe, da die ja gerade schon abgehandelt wurden).

Potential-Methode: Sei m_G die Anzahl Kanten in G. Dann definiere das Potential $\Phi = 2m_G + m_D$. Für eine Operation, sei $\Delta\Phi = \Phi_{\text{neu}} - \Phi_{\text{alt}}$ die Änderung des Potenzials, wobei Φ_{alt} das Potential vor und Φ_{neu} das Potential nach Ausführung der Operation ist. Die amortisierten Kosten sind dann die tatsächlichen Kosten plus $\Delta\Phi$.

- `clear` hat tatsächliche Kosten m_D. Außerdem verringert sich m_D auf 0 und damit ist die Potentialänderung $\Delta\Phi = -m_D$. Damit ergeben sich amortisierte Kosten 0.
- `dispose` hat tatsächliche Kosten 1. Da beim verschieben m_G um 1 schrumpft und m_D um 1 wächst verkleinert sich das Potential um 1. Also $\Delta\Phi = -1$ und `dispose` hat amortisierte Kosten 0.
- `coat` ohne Aufruf von `dispose` hat tatsächliche Kosten 1. Beim Erstellen einer Kante erhöht sich m_D um 1, daher wird das Potential um 2 größer. Also $\Delta\Phi = 2$ und `coat` hat amortisierte Kosten $3 \in \Theta(1)$.
- `coat` mit k `dispose` Aufrufen hat tatsächliche Kosten $1 + k$. Wie bei dem günstigen `coat` wird eine Kante in G erstellt, wodurch sich das Potential um 2 vergrößert. Außerdem verkleinert jeder Aufruf von `dispose` das Potential um 1. Das ergibt eine Potentialänderung von $\Delta\Phi = 2 - k$, wodurch sich amortisierte Kosten $1 + k + 2 - k = 3 \in \Theta(1)$ ergeben.