Algorithmen 1 **2. Klausur** **Wintersemester 2024/25**

31.03.2025
Lösung

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 14 Seiten (diese Titelseite eingeschlossen) und ist beidseitig bedruckt.
- Es gibt 6 Aufgaben und 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 immer asymptotisch möglichst effiziente Algorithmen erwartet.
- 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	

Page intentionally left blank.

1. **Kleinaufgaben** [10 Punkte]

(a) Ordne die folgenden Funktionen so an, dass genau dann $f \in O(g)$ gilt, wenn f links von g steht.
Hinweis: $\log(n) = \log_2(n)$ und $\log^2(n) = \log(n) \cdot \log(n)$. (3 Punkte)

$$2n \cdot \log(n), \qquad n^5, \qquad 2^{\log(n)}, \qquad 3n^2 + 2^n, \qquad (n+1)(n^2+3), \qquad \log^2(n), \qquad n\sqrt{n}$$

Lösung:

$$\log^2(n), \qquad 2^{\log(n)}, \qquad 2n \cdot \log(n), \qquad n\sqrt{n}, \qquad (n+1)(n^2+3), \qquad n^5 \qquad 3n^2 + 2^n$$

(b) Gib für die folgende Funktion $f(n)$ eine Funktion $g(n)$ an, sodass $f(n) \in \Theta(g(n))$. Vereinfache dabei die Funktion $g(n)$ so weit wie möglich. (1 Punkt)

$$f(n) = \sum_{i=0}^{\log_4 n} \left(\frac{4}{3^2}\right)^i n^2$$

Lösung: $f(n) \in \Theta(n^2)$

(c) Bestimme für folgende Situationen, welche möglichst einfache und effiziente Datenstruktur jeweils geeignet ist. Gib an, wie die geforderten Operationen und Anfragen mit der von dir gewählten Datenstruktur umgesetzt werden können. (4 Punkte)
Hinweis: In der Vorlesung haben wir folgende Datenstrukturen kennengelernt: Listen, (dynamische) Arrays, Hashtabellen, binäre Heaps, $(2,3)$-Bäume, Union-Find, Adjazenzliste.

Beispiel: Im Laden warten Personen in einer Warteschlange vor der Kasse.
i) Es können immer zwei Personen gleichzeitig bedient werden.
ii) Manchmal stellt sich eine neue Person an.
iii) Wir möchten regelmäßig wissen, welche Personen in der Warteschlange stehen.

Beispiellösung: Datenstruktur: Queue (Liste)
i) 2× Element vorne löschen
ii) Element hinten einfügen
iii) durch Liste iterieren und alle Elemente ausgeben

1. Ein paar Studierende spielen ein Spiel, bei dem man in jeder Runde Punkte bekommen kann, die zu der eigenen Punktzahl addiert werden. Je mehr Punkte, desto besser.
   i) Neue Person kommt dazu und startet mit 0 Punkten.
   ii) Person p bekommt x Punkte.
   iii) Wir möchten regelmäßig wissen: Wer ist die Person mit den wenigsten Punkten, die mehr als k Punkte hat?

   Lösung: Datenstruktur: (2,3)-Baum
   i) Element mit 0 Punkten hinzufügen
   ii) Lösche Person p und füge Person p mit neuer Punktzahl hinzu
   iii) Suche nach k Punkten, gib Nachfolger aus

2. In einer Stadt gibt es viele Häuser.
   i) Zwischen den Häusern i und j wird eine neue Straße gebaut.
   ii) Die Straße zwischen den Häusern i und j wird gesperrt.

iii) Wir möchten regelmäßig wissen, wie viele Straßen an einem Haus i liegen.

Lösung: Datenstruktur: Adjazenzliste

i) Füge Haus j in Liste von i ein und umgekehrt

ii) Lösche Haus j aus Liste von i und umgekehrt

iii) Größe der Liste i ausgeben

(d) Die folgende Funktion `magic` nimmt als Eingabeparameter ein Array mit n natürlichen Zahlen. Gib für jedes $n \in \mathbb{N}$ ein Array A mit n Zahlen an, sodass `magic`(A) Worst-Case-Laufzeit benötigt. Gib außerdem die Worst-Case-Laufzeit im O-Kalkül an. (2 Punkte)

```
Function magic(Array A):
    i := 0
    while i < n do
        if A[i] ≠ 0 then
            A[i] := 0
            i := 0
        else i := i + 1
```

Worst-Case-Instanz für $n \in \mathbb{N}$:

Lösung: $\langle 1, \ldots, 1 \rangle$

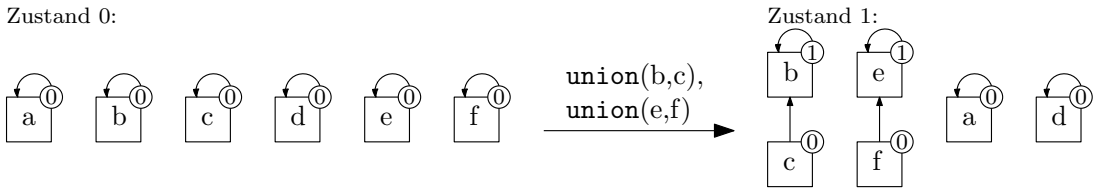
Worst-Case-Laufzeit:

Lösung: $O(n^2)$

2. **Union-Find** [10 Punkte]

In dieser Aufgabe beschäftigen wir uns mit der Union-Find-Datenstruktur, wobei wir *Union-by-Rank* und *Pfadkompression* nutzen.

Betrachte eine Union-Find Datenstruktur D, welche die Elemente a, b, c, d, e und f verwaltet. Der Rang eines Knotens steht in der oberen rechten Ecke und die Kanten verlaufen zum Elterknoten. Wir rufen zunächst nacheinander `union`(b, c) und `union`(e, f) auf und erhalten dadurch den folgenden Zustand 1.

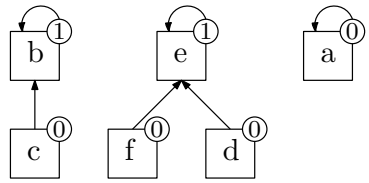


(a) Führe auf Zustand 1 die folgenden vier Operationen der Reihe nach aus und gib den Zustand der Datenstruktur *schrittweise* nach jeder dieser Operationen an. Verwende Pfadkompression und Union-by-Rank. Bei der Vereinigung von Bäumen mit Wurzeln gleichen Rangs wird die *alphabetisch kleinere* Wurzel als neue Wurzel ausgewählt. (4 Punkte)

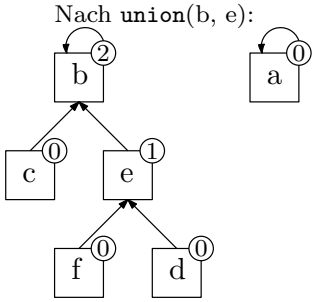
`union`(e, d), `union`(b, e), `find`(d), `union`(a, f)

Lösung:

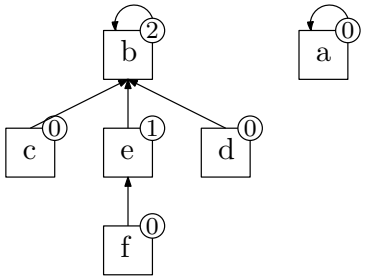
Nach `union`(e, d):



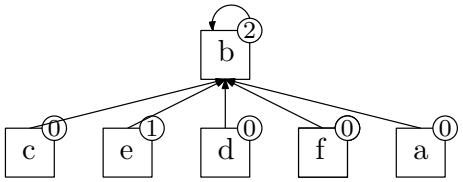
Nach `union`(b, e):



Nach `find`(d):



Nach `union`(a,f):



(b) Betrachte eine Union-Find-Datenstruktur mit n Elementen, die wir von 0 bis $n-1$ durchnummerieren. Die Datenstruktur besteht aus zwei Arrays der Größe n:

- `rank`, wobei `rank`$[i]$ den Rang von Element i speichert.
- `parent`, wobei `parent`$[i]$ den Elter von Element i speichert. Wenn Element i eine Wurzel ist, dann ist `parent`$[i] = i$.

Außerdem ist die Funktion `find`(i) gegeben, welche die Wurzel der Menge, zu der Element i gehört, zurückgibt. Gib Pseudocode an, der die `union`-Operation mit Union-by-Rank für zwei gegebene Elemente $a, b \in \{0, \dots, n-1\}$ umsetzt. (4 Punkte)

Function `union`$(a\colon \mathbb{N}, b\colon \mathbb{N})$**:**

Lösung:
$x \coloneqq$ `find`(a)
$y \coloneqq$ `find`(b)
if $x = y$ **then return**
if *rank[y]* $<$ *rank[x]* **then** `parent`[y] $\coloneqq$ x
else if *rank[y]* $>$ *rank[x]* **then** `parent`[x] $\coloneqq$ y
else
  `parent`[y] $\coloneqq$ x
  `rank`[x] $+\!=$ 1

(c) Betrachte eine Union-Find-Datenstruktur mit Elementen in $\mathbb{N}$. Wir verwenden nun Union-Find mit Pfadkompression, aber *ohne* Union-By-Rank. Stattdessen wird bei der Vereinigung immer die Wurzel mit dem *größeren* Index als neue Wurzel ausgewählt.

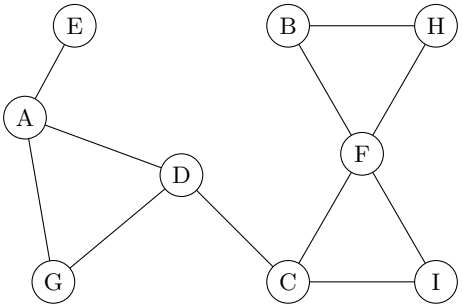
Ausgehend von einer leeren Datenstruktur, gib für jedes $n \in \mathbb{N}$ eine Sequenz von $O(n)$ `union`-Operationen und einer `find`-Operation an, sodass die `find`-Operation $\Theta(n)$ Laufzeit benötigt. Begründe deine Antwort kurz. (2 Punkte)

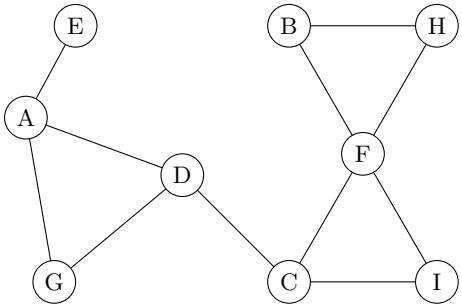
Lösung: `union`$(0, 1)$, `union`$(1, 2)$, `union`$(n-2, n-1)$, $\dots$, `find`(0). Die Elemente, auf den Union aufgerufen wird, sind immer Wurzeln. Dabei ist die Menge des größeren Elements ein isolierter Knoten und die Menge des kleineren Elements ein Pfad. Dieser Pfad wird an das größere Element gehängt. Nach $n-1$ `union`-Operationen erhalten wir einen Pfad der Länge n. Dann benötigt `find`(0) $\Theta(n)$ Zeit, um die Wurzel $n-1$ zu finden.

3. **DFS + Brücken** [10 Punkte]

(a) Führe in folgendem ungerichteten Graphen eine Tiefensuche von A aus und zeichne den DFS-Baum, indem du dessen Kanten in der Darstellung markierst oder sie angibst. Trage außerdem die entstehenden DFS- und Low-Werte für jeden Knoten ein, wie für den Beispiel-Knoten A angegeben. Falls zu einem Zeitpunkt mehrere unbesuchte Nachbarn zur Auswahl stehen, dann besuche den alphabetisch kleineren Knoten zuerst.

Zur Erinnerung: Der Low-Wert $\text{low}(v)$ eines Knotens v ist das Minimum aus der DFS-Nummer von v und der kleinsten DFS-Nummer, die man von einem Knoten im Teilbaum unter v mit einer Rückkante erreichen kann. (4 Punkte)

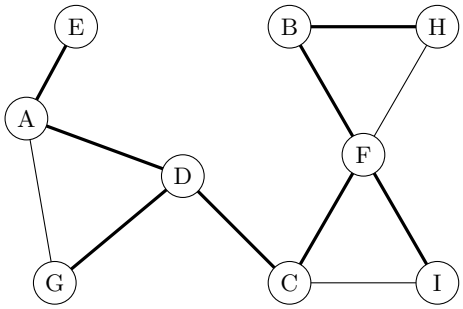




Kopie desselben Graphen, falls du dich beim ersten vertan hast. Markiere deutlich, welche Lösung zu bewerten ist.

Knoten	A	B	C	D	E	F	G	H	I
dfs	0								
low	0								

Lösung:



Knoten	A	B	C	D	E	F	G	H	I
dfs	0	4	2	1	8	3	7	5	6
low	0	3	2	0	8	2	0	3	2

(b) Gib für das obige Beispiel alle Brücken und alle Cut-Vertices an. (2 Punkte)

Zur Erinnerung: In einem zusammenhängenden Graph ist ein Knoten v ein Cut-Vertex, wenn der Graph nach dem Löschen von v nicht mehr zusammenhängend ist.

Brücken:

Lösung: AE, DC

Cut-Vertices:

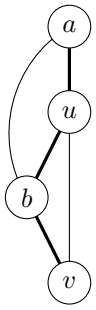
Lösung: A, D, F, C

(c) Zeige oder widerlege die folgende Aussage. Falls es eine DFS-Traversierung eines ungerichteten Graphen G gibt, bei der keine Rückkanten gefunden werden, dann ist jede Kante in G eine Brücke. (2 Punkte)

Lösung: Aussage stimmt. Wenn keine Rückkanten gefunden werden, ist der Graph kreisfrei und jede Kante ist eine Brücke.

(d) Widerlege die folgende Aussage. Seien u und v zwei adjazente Knoten in einem ungerichteten Graphen G. Wenn es eine DFS-Traversierung gibt, bei der sich die low-Werte von u und v unterscheiden, dann ist $\{u, v\}$ eine Brücke. (2 Punkte)

Lösung: Die Aussage ist falsch. Im folgenden DFS-Baum mit Wurzel a ergeben sich für u und v verschiedene low-Werte, obwohl $\{u, v\}$ keine Brücke ist.



Knoten	a	u	b	v
dfs	0	1	2	3
low	0	0	0	1

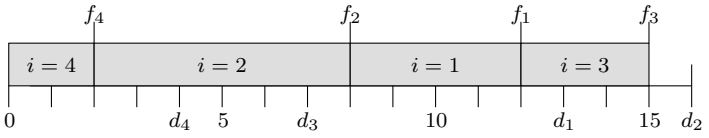
4. **Zeitplan** [10 Punkte]

Sophie hat sehr viele Aufgaben, die sie zu einer bestimmten Zeit erledigt haben muss. Sie macht jede Aufgabe direkt nach der anderen (in einer beliebigen Reihenfolge). Leider hat sie so viel zu tun, dass es sein kann, dass eine Aufgabe nicht rechtzeitig fertig wird. Sophie möchte die Reihenfolge ihrer Aufgaben so wählen, sodass (falls möglich) alle Aufgaben rechtzeitig fertig werden.

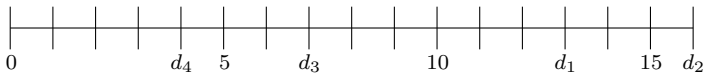
Bei dem Problem ZEITPLAN gibt es als Eingabe n Aufgaben $A = \{1, \ldots, n\}$, wobei Aufgabe i insgesamt t_i Stunden braucht und eine Deadline d_i hat. Die Lösung des Problems besteht aus einer *gültigen* Reihenfolge aller Aufgaben. Eine Reihenfolge ist gültig, wenn für alle Aufgaben i gilt, dass $f_i \leq d_i$, wobei f_i die Zeit ist, zu der Aufgabe i fertig wird.

Im folgenden Beispiel mit vier Aufgaben haben wir, wie rechts dargestellt, die Reihenfolge $4, 2, 1, 3$ gewählt. Damit werden alle Aufgaben rechtzeitig fertig, außer Aufgabe 3, da $f_3 > d_3$.

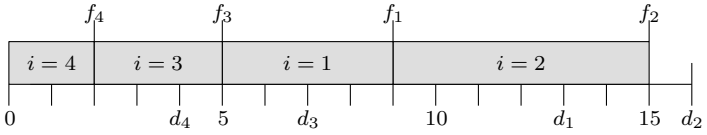
Aufgabe i	1	2	3	4
Dauer t_i	4	6	3	2
Deadline d_i	13	16	7	4



(a) Gib eine gültige Reihenfolge für die Probleminstanz aus dem obigen Beispiel an. Zeichne dazu die Aufgaben in der gültigen Reihenfolge ein oder gib die Reihenfolge der Aufgaben an. (2 Punkte)



Lösung:



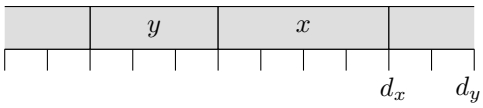
(b) Sophie möchte ZEITPLAN nun lösen, indem sie die Aufgaben aufsteigend nach ihrer Dauer sortiert. Gib eine Eingabe an, bei der mit dieser Vorgehensweise nicht alle Aufgaben rechtzeitig fertig werden, obwohl es eine gültige Reihenfolge gibt. Gib zusätzlich für dein Beispiel auch eine Reihenfolge an, bei der alle Aufgaben rechtzeitig fertig werden. (2 Punkte)

Lösung: Betrachte das folgende Beispiel:

Aufgabe i	1	2
Dauer t_i	1	2
Deadline d_i	3	2

Wenn wir nach der Dauer sortieren, wird Aufgabe 2 nicht rechtzeitig fertig. Alle Aufgaben werden rechtzeitig fertig, wenn Sophie zuerst Aufgabe 2 und dann Aufgabe 1 macht.

(c) Sei für ein ZEITPLAN Problem eine gültige Reihenfolge gegeben, das heißt, alle Aufgaben werden rechtzeitig fertig. Wir nehmen an, dass es zwei Aufgaben x und y gibt, wobei Sophie Aufgabe x direkt *nach* Aufgabe y macht, die Deadline von x aber *vor* y ist. Es gilt also $d_x < d_y$. Das untere Beispiel zeigt solch eine mögliche Situation.



Zeige, dass die Reihenfolge gültig bleibt, wenn *nur* die Reihenfolge der Aufgaben x und y vertauscht wird. Die Reihenfolge der anderen Aufgaben ändert sich nicht. (2 Punkte)

Lösung: Nach Voraussetzung gilt für die ursprüngliche Reihenfolge $f_y < f_x \leq d_x < d_y$. Aufgabe x wird nach dem Tausch früher fertig, also gilt $f'_x \leq d_x$. Aufgabe y wird nach dem Tausch zum Zeitpunkt $f'_y = f_x \leq d_x$ fertig. Wegen $d_x < d_y$ sind also beide Aufgaben rechtzeitig fertig. Für die anderen Aufgaben ändert sich nicht, wann sie fertig werden. Daher bleibt die Reihenfolge nach dem Tausch gültig.

(d) Beschreibe einen Algorithmus in Worten, der für eine gegebene ZEITPLAN-Instanz eine gültige Reihenfolge berechnet, falls diese existiert. Begründe die Korrektheit deines Algorithmus und gib außerdem die Laufzeit an. (4 Punkte)

Lösung: Sortiere die Aufgaben aufsteigend nach ihrer Deadline und gebe die Aufgaben in dieser Reihenfolge aus. Das Sortieren benötigt $O(n \log(n))$ Zeit.

Korrektheit: Ist eine gültige Reihenfolge nicht nach Deadline sortiert, dann gibt es zwei aufeinanderfolgende Aufgaben, bei der die Aufgabe mit der späteren Deadline zuerst gemacht wird. Aus Teilaufgabe (c) wissen wir, dass wir diese beiden Aufgaben tauschen können, wobei die Reihenfolge gültig bleibt. Dadurch wird die Anzahl der vertauschten Aufgaben echt kleiner. Dieses Argument können wir weiterführen bis alle Aufgaben sortiert sind.

Wenn es eine gültige Reihenfolge gibt, gibt es also immer auch eine gültige Reihenfolge, bei der alle Aufgaben nach ihrer Deadline sortiert sind.

5. **Bücherregal** [10 Punkte]

Ada hat ein Bücherregal, das $k \in \mathbb{N}$ Zentimeter breit ist. Insgesamt hat sie n Bücher $B = \{1, \ldots, n\}$, wobei Buch i eine Breite von $w_i \in \mathbb{N}$ Zentimetern hat.

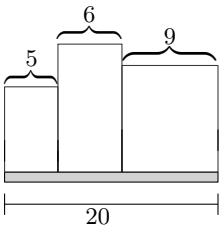
Sie möchte ihr Bücherregal mit möglichst vielen Büchern *vollständig* füllen. Gesucht ist also eine Teilmenge $S \subseteq B$ von Büchern, sodass die Summe der Breiten aller Bücher in S *genau* k ist und $|S|$ maximal ist. Das Problem BÜCHERREGAL fragt bei Eingabe von $W = (w_1, \ldots, w_n)$ und k nach der maximalen Größe $|S|$ einer solchen Teilmenge.

Für $W = (5, 6, 3)$ und $k = 10$ kann Ada *nie* ihr Bücherregal vollständig füllen, egal welche Bücher sie wählt. Dieses Beispiel ist also *nicht lösbar*.

Das folgende Beispiel ist *lösbar*.

$k = 20$

Buch i	1	2	3	4	5	6
Breite w_i	1	3	5	6	9	4
Ins Regal?	○	○	⊗	⊗	⊗	○



Die drei markierten Bücher sind insgesamt genau 20 Zentimeter breit (siehe Grafik rechts). Das ist aber *nicht optimal*, da sie ihr Bücherregal mit mehr als drei Büchern vollständig füllen kann.

(a) Zeige, dass die markierte Teilmenge im obigen Beispiel nicht optimal ist, indem du eine optimale Teilmenge angibst (bspw. durch Markieren in der unteren Tabelle). (2 Punkte)

$k = 20$

Buch i	1	2	3	4	5	6
Breite w_i	1	3	5	6	9	4
Ins Regal?	○	○	○	○	○	○

Lösung:

Buch i	1	2	3	4	5	6
Breite w_i	1	3	5	6	9	4
Ins Regal?	⊗	○	○	⊗	⊗	⊗

Ada kann mit insgesamt vier Büchern ihr Bücherregal vollständig füllen.

(b) Angenommen, Ada weiß schon für jede Breite $b \leq k$, ob die Instanz mit Breite b und den Büchern $\{1, \ldots, n-1\}$ lösbar ist. Beschreibe, wie sie bestimmen kann, ob die Instanz mit Breite k und den Büchern $\{1, \ldots, n\}$ lösbar ist. (2 Punkte)

Lösung: Wenn die Instanz mit Breite k und $n-1$ Büchern lösbar ist, dann auch die Instanz mit Breite k und n Büchern. Ansonsten ist die Instanz mit Breite k und n Büchern genau dann lösbar, wenn die Instanz mit Breite $k - w_n$ und $n-1$ Büchern lösbar ist.

Wir wollen nun ein dynamisches Programm für BÜCHERREGAL entwickeln, wobei wir zuerst nur für eine Teilmenge von Büchern und ein kleineres Bücherregal die optimale Lösung bestimmen wollen. Das DP soll für jeden Index $i \in \{0, \dots, n\}$ und jede Breite $b \in \{0, \dots k\}$ den Wert einer optimalen Teillösung $X[i][b]$ in einem Array verwalten. Falls eine Teilinstanz nicht lösbar ist, weisen wir ihr den Wert $-\infty$ zu.

(c) Gib an, welche Bedeutung eine Teillösung $X[i][b]$ 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.

Lösung: $X[i][b]$ entspricht der maximalen Anzahl an Büchern in $\{1, \dots, i\}$, mit denen ein Bücherregal der Breite b vollständig gefüllt werden kann.
Für $i = 0$: $X[0][0] \coloneqq 0$ und $X[0][b] = -\infty$ für $b > 0$. Für $i > 0$:

$$X[i][b] = \begin{cases} \max\{X[i-1][b], 1 + X[i-1][b - w_i]\}, & \text{falls } w_i \leq b \\ X[i-1][b] & \text{sonst.} \end{cases}$$

(d) Gib an, wie anhand der ausgefüllten DP-Tabelle bestimmt werden kann, ob es eine Lösung gibt oder nicht. (1 Punkt)

Lösung: Die Instanz ist genau dann lösbar, wenn $X[n][k] \neq -\infty$.

6. **Sortiert-unsortierte Folge** [10 Punkte]

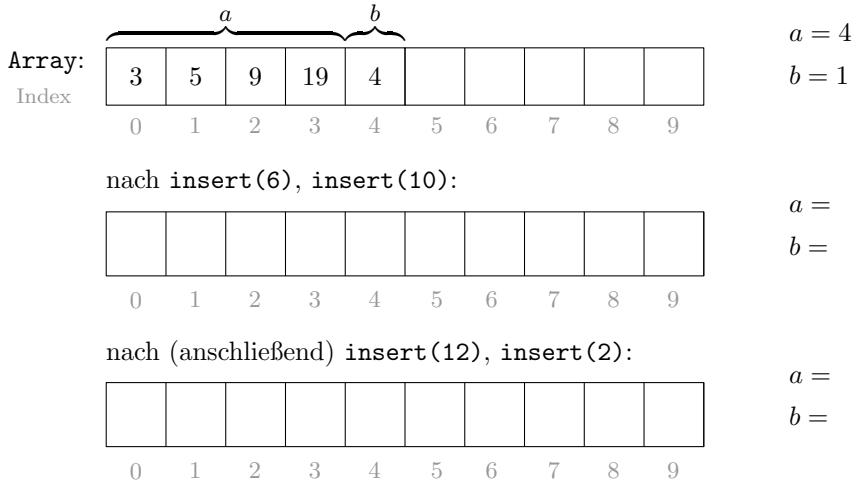
Wir wollen eine sortierte Folge mittels eines dynamischen Arrays A implementieren. Dabei soll folgende Invariante aufrechterhalten bleiben: A enthält $n = a + b$ Zahlen mit $b \leq \sqrt{a}$, wobei die ersten a Zahlen sortiert sind und die restlichen b Zahlen nicht notwendigerweise sortiert sind.

Die Datenstruktur wird folgende Operationen unterstützen:

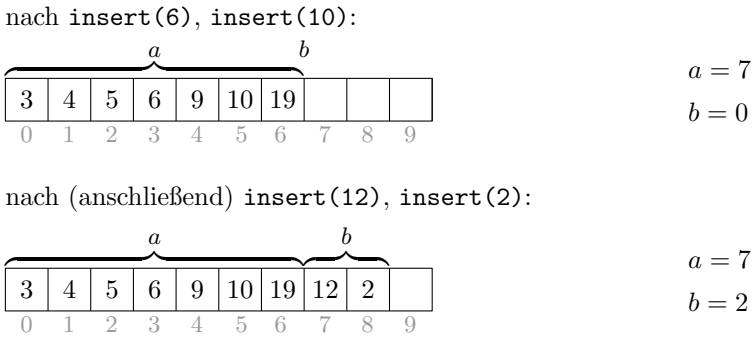
- `insert(e)` Fügt eine Zahl in die Datenstruktur ein.
- `find(e)` Gibt die kleinste Zahl in der Datenstruktur zurück, die größer gleich `e` ist. Falls es keine solche Zahl gibt, gibt `find(e)` ∞ zurück.

Die Operation `insert(e)` ist wie folgt umgesetzt: Für jeden Aufruf von `insert(e)` wird `pushBack(e)` auf A ausgeführt (dies erhöht b um eins). Falls die Invariante nun nicht mehr erfüllt ist, also $b > \sqrt{a}$ gilt, wird das gesamte Array sortiert, sodass anschließend $a = n$ und $b = 0$ gilt.

(a) Betrachte die folgende, schon teilweise gefüllte Datenstruktur mit $a = 4$ und $b = 1$. Auf der Datenstruktur werden nun vier `insert` Operationen ausgeführt. Gib für das untere Beispiel den Zustand der Datenstruktur nach den Operationen `insert(6)`, `insert(10)` und nach den Operationen `insert(12)`, `insert(2)` an. Gib außerdem jeweils a und b an. (2 Punkte)



Lösung:



(b) Beschreibe, wie du auf der Datenstruktur `find(e)` mit Laufzeit in $O(\log(a)+b)$ ausführen kannst. Begründe, warum deine Methode die angegebene Laufzeit besitzt. (3 Punkte)

Lösung: `find(e)` führt auf den ersten a Zahlen eine binäre Suche aus, um das erste Element größer oder gleich `e` zu finden. Dies ist in $O(\log a)$ Zeit möglich. In den letzten b Einträgen wird mittels linearer Suche das erste Elemente größer oder gleich `e` gesucht. Von den beiden gefundenen Elementen nehmen wir das Minimum. Damit ist die Laufzeit insgesamt $O(\log a + b)$.

Die Operation `find(e)` hat also eine Laufzeit von $O(\log(a) + b)$. Die Operation `insert(e)` hat Laufzeit $O(1)$, falls A nicht sortiert werden muss. Falls A sortiert werden muss, nehmen wir vereinfachend an, dass `insert(e)` eine Laufzeit von $O(n)$ hat.

(c) Zeige, dass ausgehend von einer leeren Datenstruktur die amortisierten Kosten von `insert(e)` und `find(e)` in $O(\sqrt{n})$ liegen. (5 Punkte)

Lösung: Wir zeigen zunächst, dass $b \in \Theta(\sqrt{n})$ gilt, falls bei einem `insert(e)` neu sortiert werden muss. Für die untere Schranke gilt $b > \sqrt{a}$ (andernfalls müsste nicht sortiert werden) und außerdem $a + b = n$. Somit erhalten wir

$$\begin{aligned} & b > \sqrt{n-b} \\ \Longrightarrow & b^2 + b > n \\ \Longrightarrow & 2b^2 > n \\ \Longrightarrow & b \in \Omega(\sqrt{n}). \end{aligned}$$

Die obere Schranke folgt, da $b \leq \sqrt{a} \leq \sqrt{n}$.

Charging

Wir chargen die Kosten von teuren `insert(e)` Operationen auf vorherige günstige `insert(e)` Operationen. Es können zwei Fälle auftreten:

- Nach dem Aufruf von `insert(e)` wird A nicht sortiert. Die Operation verursacht dann $O(1)$ Kostentoken. Diese müssen nicht verteilt werden.
- Nach dem Aufruf von `insert(e)` wird A sortiert. In diesem Fall entstehen $O(n)$ Kostentoken. Die Kostentoken werden gleichmäßig auf die b `insert` Operationen verteilt, welche Elemente in den unsortierten Teil eingefügt haben. Weil $b \in \Omega(\sqrt{n})$, werden jeder dieser Operation $O(\sqrt{n})$ Kostentoken zugewiesen. Da Elemente nur einmal aus dem unsortierten in den sortierten Bereich geschoben werden, werden `insert` Operationen nur einmal gecharged.

Jede `insert` Operation wird also mit $O(\sqrt{n})$ Kostentoken gecharged.

Da `find(e)` keine Tokens erhält, sind die amortisierten Kosten auch gleich den worst-case Kosten. Weil $a \leq n$ und $b < \sqrt{a} \leq \sqrt{n}$, sind diese in $O(\log a + b) = O(\sqrt{n})$.

Kontomethode

Die `insert(e)` Operationen zahlen jeweils $\sqrt{n}$ Tokens auf das Konto ein. Falls die Elemente nicht sortiert werden, werden nur Tokens eingezahlt. Andernfalls entstehen $O(n)$ Kosten, welche wir mit dem Konto ausgleichen müssen. Auf dem Konto wurden während der letzten b Operationen insgesamt $\sqrt{n}+\sqrt{n-1}+\cdots+\sqrt{n-b+1} \geq b \cdot \sqrt{n-b+1}$ Tokens eingezahlt. Da $b \in \Theta(\sqrt{n})$ ist, gilt $b \cdot \sqrt{n-b+1} \in \Theta(n)$. Damit sind genug Tokens auf dem Konto vorhanden, um die Kosten zu decken.

Die `find(e)` Operationen zahlen nichts auf das Konto ein und haben somit $O(\sqrt{n})$ Kosten, analog zur Argumentation bei der Charging-Methode.

Potentialmethode

Wir definieren unser Potential als $\Phi = b \cdot \sqrt{n}$, wobei n die aktuelle Gesamtzahl an Elementen und b die Anzahl der unsortierten Elemente ist. Dann gilt für die amortisierten Kosten:

- `insert(e)` ohne sortieren hat tatsächliche Kosten von $O(1)$. Sowohl n als auch b erhöhen sich um 1. Es ist $\Phi_{\text{neu}} - \Phi_{\text{alt}} = (b+1)\sqrt{n+1} - b\sqrt{n} = \sqrt{n+1} + b(\sqrt{n+1} - \sqrt{n}) \leq \sqrt{n+1} + b \in O(\sqrt{n})$. Insgesamt sind die Kosten damit in $O(\sqrt{n})$.
- `insert(e)` mit sortieren hat tatsächliche Kosten von $O(n)$. Dabei erhöht sich n um 1 und b wird auf 0 gesetzt. Es ist $\Phi_{\text{neu}} - \Phi_{\text{alt}} = 0 \cdot \sqrt{n+1} - b\sqrt{n} = -b\sqrt{n}$. Mit $b \in \Theta(\sqrt{n})$ folgt $\Phi_{\text{alt}} - \Phi_{\text{neu}} \in O(n)$. Die gesamten Kosten sind also in $O(1) \subseteq O(\sqrt{n})$.
- Die `find(e)` Operationen verändern das Potential nicht und haben tatsächliche Kosten von $O(\sqrt{n})$ mit dem Argument aus der Charging-Methode.