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

01.03.2024

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 17 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 $f \in O(g)$ gilt, wenn f links von g steht. (3 Punkte)

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

Lösung:

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

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

Laufzeit:

Lösung:

$$f(n) = n \log(n)$$

(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 Anzahl der Zusammenhangskomponenten in einem ungerichteten Graphen mit n Knoten und m Kanten. Der Graph ist als Adjazenzliste gegeben.

   Lösung: $O(n + m)$

2. Bestimme, ob ein sortiertes Array mit n Zahlen eine gegebene Zahl enthält.

   Lösung: $O(\log(n))$

3. Rotiere ein Array mit n Elementen um 5 Stellen. (Das heißt, der Eintrag an Index i soll danach an Index $(i + 5) \bmod n$ stehen).

   Lösung: $O(n)$

(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. Die Teilnehmenden eines Wettbewerbs sollen mit einem Sandsäckchen möglichst nah an ein Ziel werfen. Sie haben beliebig viele Versuche, um die Entfernung zum Zielpunkt zu minimieren, wobei immer nur der beste Versuch pro Teilnehmer:in gespeichert wird. Wer den Zielpunkt genau getroffen hat, wirft nicht nochmal. Die Jury möchte nach jedem Wurf wissen, wer bisher den besten Versuch hatte, aber den Zielpunkt noch nicht getroffen hat.

   Datenstruktur:

   Lösung: Heap

   Operationen:

   Lösung: Einfügen, kleinstes Element löschen, kleinstes Element bestimmen

2. Edsger bekommt jeden Tag zwei neue Gartenzwerge (einen kleinen und einen großen) von Ada geschenkt. Begeistert möchte er Ada immer erzählen, welche alten Gartenzwerge von der Größe her zwischen den Größen der beiden neuen Gartenzwerge liegen.

   Datenstruktur:

   Lösung: (2, 3)-Baum

   Operationen:

   Lösung: Einfügen, Suchen

3. Eine Lesegruppe verwaltet einen Turm von Büchern, wobei das oberste Buch zum Lesen weggenommen oder darauf ein neues Buch gelegt werden kann. Zu jedem Zeitpunkt möchten die Mitglieder der Gruppe bestimmen können, welches Buch aktuell genommen werden kann.

   Datenstruktur:

   Lösung: Dynamisches Array (Stack)

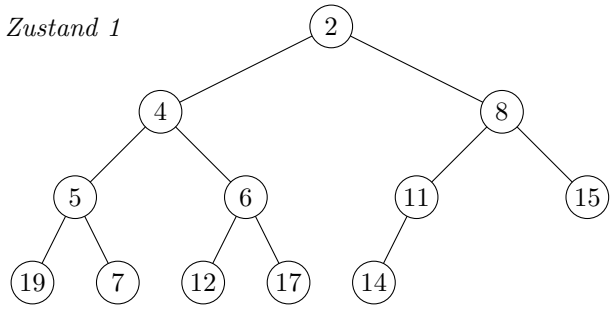
   Operationen:

   Lösung: Hinten einfügen, letztes Element bestimmen, letztes Element löschen.

2. **Heaps** [10 Punkte]

In dieser Aufgabe beschäftigen wir uns mit Min-Heaps.

(a) Gegeben sei folgender Min-Heap in *Zustand 1* zusammen mit der rechts angegebenen Sequenz von Operationen, die diesen Heap schrittweise in verschiedene Zustände überführt.

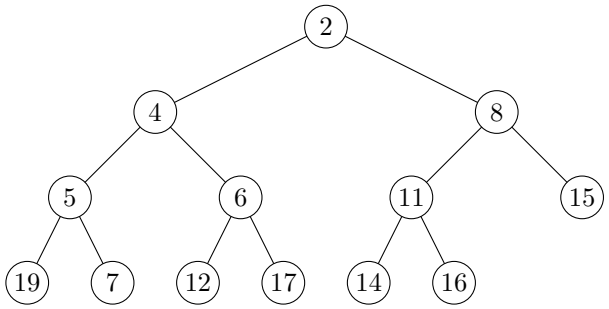


Zustand 1
`push(16)`
Zustand 2
`popMin()`
Zustand 3
`push(7)`
Zustand 4

Zeichne die so entstehenden Zustände 2, 3 und 4 des Heaps. (4 Punkte)

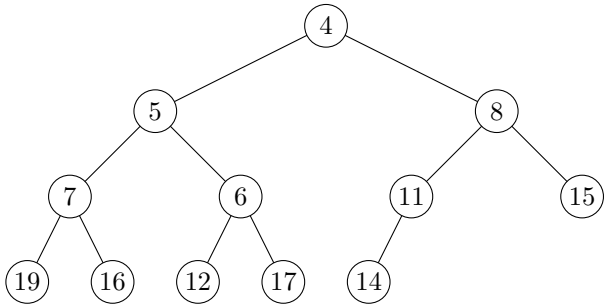
Zustand 2:

Lösung:



Zustand 3:

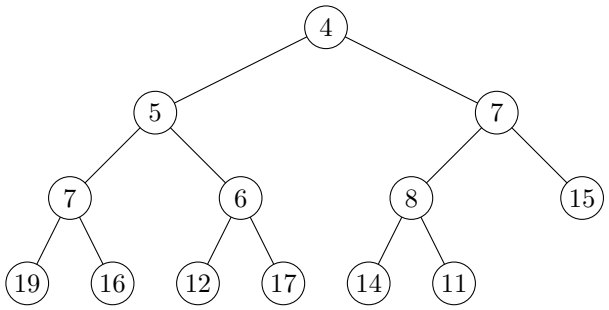
Lösung:



Zeichne Zustand 4 oben auf der nächsten Seite.

Zustand 4:

Lösung:



(b) In der Vorlesung haben wir gelernt, wie Min-Heaps als Arrays umgesetzt werden können. Gib das Array A an, welches den Min-Heap in *Zustand 1* aus Teilaufgabe a) repräsentiert. (1 Punkt)

$A =$ | | | | | | | | | | | | |

Lösung:

$A =$ | 2 | 4 | 8 | 5 | 6 | 11 | 15 | 19 | 7 | 12 | 17 | 14 |

(c) Sei `H` ein Min-Heap mit n Elementen. Im Folgenden soll eine Methode `reduceToLargerHalf(H)` implementiert werden, welche die $\lfloor n/2 \rfloor$ kleinsten Elemente aus `H` in sortierter Reihenfolge ausgibt, indem sie diese aus `H` löscht. Dazu stehen, neben den Heap-Operationen, die aus der Vorlesung bekannt sind, zusätzlich noch folgende Operationen zur Verfügung:

- `H.size()` gibt die Anzahl der Elemente in `H` zurück.
- `print(`x`)` gibt den Wert des Elements x aus.

Gib den Pseudocode für `reduceToLargerHalf(H)` an. (2 Punkte)

`reduceToLargerHalf`(`H`: HEAP) :

Lösung:

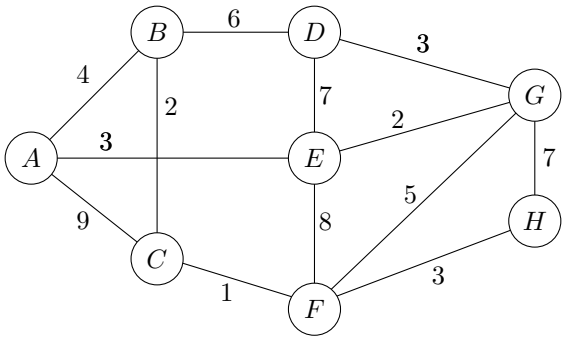
```
k := ⌊H.size()/2⌋
for i from 1 to k do
    e := H.popMin()
    print(e)
```

(d) Gegeben sei ein leerer Heap. Zeige oder widerlege: Für jedes $n \in \mathbb{N}$ existiert eine Folge von n `insert`-Operationen, sodass jede dieser `insert`-Operationen Laufzeit $\Theta(1)$ hat.
Begründe deine Antwort kurz. (3 Punkte)

Lösung: Die Aussage ist wahr. Wir wählen $A = \langle 1, 2, \ldots, n \rangle$. Bei `insert`$(a_i)$ enthält der Heap nur die Elemente $a_1, \ldots, a_{i-1}$ und ein neues Blatt a_i wird unten angehängt. Da $a_i > a_j$ für alle $j < i$, ist die Priorität des Elters von a_i größer als die von a_i selbst. Da dies als Invariante für alle `insert`-Operationen gilt, bleibt dabei die Heap-Eigenschaft erhalten. Somit wird kein Bubble-Up durchgeführt und die Laufzeit der Operation beträgt $\Theta(1)$.

3. **Dijkstras Algorithmus** [10 Punkte]

Im Folgenden betrachten wir gewichtete Graphen $G = (V, E)$ mit Kantengewichtsfunktion $w\colon E \to \mathbb{N}$.
Wir wollen nun Dijkstras Algorithmus auf dem folgenden Graphen mit Startknoten A anwenden.

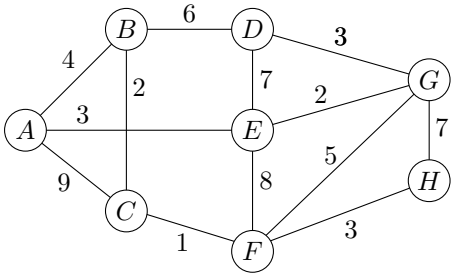


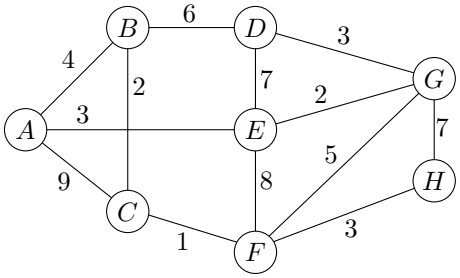
(a) Gib die Reihenfolge an, in der die Knoten aus der Priority-Queue entnommen werden. Falls es in einem Schritt mehrere Knoten gibt, die in Frage kommen, wähle den alphabetisch kleineren Knoten. Markiere außerdem die Kanten des zugehörigen Kürzeste-Wege-Baums in einer der unteren Kopien.
Hinweis: Der Kürzeste-Wege-Baum ist ein Baum, der für jeden Knoten v den kürzesten Weg vom Startknoten zu v enthält. (4 Punkte)

Reihenfolge, in der die Knoten aus der Priority-Queue entnommen werden:

Lösung: A, E, B, G, C, F, D, H

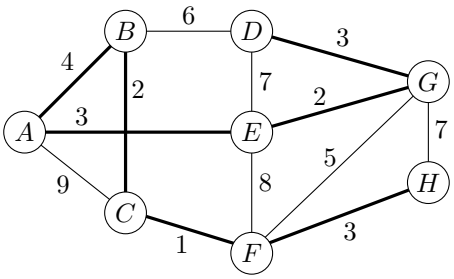
Kürzester-Wege-Baum:





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

Lösung:



(b) Wir betrachten nun das Distanzarray D, welches bei der Ausführung von Dijkstras Algorithmus auf dem obigen Graph zu jedem Zeitpunkt die Länge des aktuell bekannten kürzesten Weges von A zu allen Knoten in V speichert. Im Folgenden sind Zustände von D angegeben, die während einer solchen Ausführung *nicht* auftreten können. Insbesondere ist *genau ein* Eintrag fehlerhaft, der in den Tabellen jeweils markiert ist. In folgendem Beispiel ist der Eintrag von A fehlerhaft, weil die Distanz des Startknotens mit 0 initialisiert wird und nicht mit 2:

Knoten	$\boxed{A}$	B	C	D	E	F	G	H
aktuelle Distanz zu A	$\boxed{2}$	∞	∞	∞	∞	∞	∞	∞

Gib für die folgenden beiden fehlerhaften Zustände jeweils eine Begründung an, warum der markierte Eintrag falsch ist. (3 Punkte)

1.

Knoten	A	B	C	D	E	F	G	$\boxed{H}$
aktuelle Distanz zu A	0	4	9	∞	3	∞	∞	$\boxed{12}$

Lösung: Es wurde noch kein Nachbar von H exploriert, daher kann H noch nicht entdeckt worden sein.

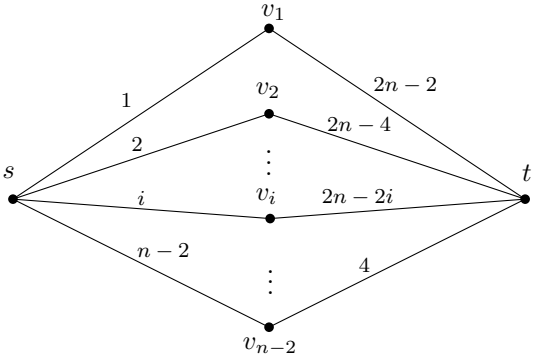
2.

Knoten	A	B	C	$\boxed{D}$	E	F	G	H
aktuelle Distanz zu A	0	4	6	$\boxed{11}$	3	10	5	12

Lösung: Knoten D wird zum ersten Mal von E aus entdeckt, wobei die aktuelle Distanz auf 10 gesetzt wird. Da Distanzen während der Ausführung nur kleiner werden können, kann die aktuelle Distanz von E zu A zu keinem Zeitpunkt 11 sein.

(c) Gib für jedes $n > 3$ einen Graphen mit n Knoten sowie zwei Knoten s und t an, sodass sich bei der Ausführung von Dijkstras Algorithmus mit Startknoten s die Länge des aktuell bekannten kürzesten Weges von s zu t insgesamt $\Theta(n)$ Mal ändert. Begründe deine Antwort kurz. (3 Punkte)

Lösung: Für ein $n > 3$ sieht der Graph wie folgt aus:



Die Knotenmenge besteht aus $V = \{s, t, v_1, \ldots, v_{n-2}\}$, wobei s und t jeweils adjazent zu allen Knoten in $\{v_1, \ldots, v_{n-2}\}$. Außerdem ist $w(\{s, v_i\}) = i$ und $w(\{t, v_i\}) = 2n - 2i$ für alle $i \in \{1, \ldots, n-2\}$.

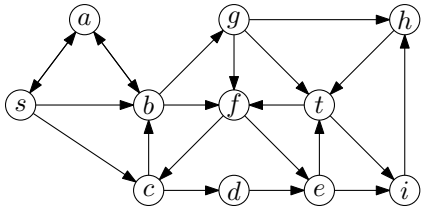
Zuerst werden alle Knoten v_i von s aus entdeckt und in aufsteigender Reihenfolge exploriert. Dabei wird jeweils die Kante $\{v_i, t\}$ betrachtet. Da $\text{dist}(s, v_i) + \text{dist}(v_i, t) = i + 2n - 2i = 2n - i$ echt kleiner wird für echt größeres i, wird die aktuelle Distanz von s zu v jedes Mal aktualisiert, wenn ein Knoten v_i exploriert wird. Das sind insgesamt dann $n - 3 \in \Theta(n)$ Änderungen.

4. **Via-Knoten** [10 Punkte]

Gegeben sei ein gerichteter Graph $G = (V, E)$, zwei Knoten $s, t \in V$ sowie zwei natürliche Zahlen $d_s, d_t \in \mathbb{N}$. Wir nennen $v \in V$ einen *Via-Knoten*, wenn $\text{dist}(s, v) = d_s$ und $\text{dist}(v, t) = d_t$ gilt, wobei $\text{dist}(x, y)$ die Distanz von Knoten x zu y ist. Das heißt, ein Via-Knoten ist ein Knoten, der Distanz d_s von s und Distanz d_t zu t hat.

Im Folgenden möchten wir einen Algorithmus entwickeln, der alle Via-Knoten des Graphen berechnet.

(a) Gegeben sei folgender Graph zusammen mit $d_s = 1$ und $d_t = 3$. Markiere alle Via-Knoten oder gib sie an. (1 Punkt)



Lösung: Knoten a und c

(b) Gib Werte für d_s und d_t an, sodass im Graphen aus Teilaufgabe a) Knoten f ein Via-Knoten ist.(1 Punkt)

$d_s =$ $d_t =$

Lösung: $d_s = 2$, $d_t = 2$

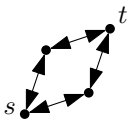
(c) Sei G ein gerichteter Graph, seien $s, t \in V(G)$ und $d_s, d_t \in \mathbb{N}$. Entscheide für jede der *drei* folgenden Aussagen, ob sie wahr oder falsch ist und begründe deine Antwort kurz. (3 Punkte)

1. Falls $d_s + d_t < \text{dist}(s, t)$, kann es keinen Via-Knoten geben.

   Lösung: Aussage stimmt. Angenommen, es gäbe einen Via-Knoten v, dann ist der kürzeste Pfad von s zu v zusammen mit dem von v zu t auch einen s-t-Pfad, der kürzer ist als $\text{dist}(s, v)$. Dies ist ein Widerspruch.

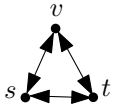
2. Falls $d_s + d_t = \text{dist}(s, t)$, gibt es immer genau einen Via-Knoten.

   Lösung: Nein, siehe Gegenbeispiel mit $d_s = d_t = 1$:

3. Falls $d_t \leq \text{dist}(s, t)$ ist, dann liegen alle Via-Knoten auf einem kürzesten s-t-Pfad.

   Lösung: Nein, siehe Gegenbeispiel mit $d_s = d_t = 1 \leq \text{dist}(s, t)$:

   Knoten v ist der einzige Via-Knoten, liegt aber nicht auf einem kürzesten Pfad.

Im Folgenden seien $n = |V|$ und $m = |E|$.

(d) Beschreibe, wie man in einem gerichteten Graphen mit n Knoten und m Kanten in $O(n + m)$ Zeit die Distanz von allen Knoten zu einem gegebenen Knoten x berechnen kann. (2 Punkte)

Lösung: Wir berechnen den invertierten Graphen G^{-1}, den wir aus G erhalten, indem wir die Richtung jeder Kante umdrehen. Anschließend können wir mit einer Breitensuche von x aus alle Distanzen von x in G^{-1} berechnen. Dies sind dann genau die Distanzen *zu* x in G.

(e) Beschreibe einen Algorithmus in Worten, der als Eingabe einen gerichteten Graphen, zwei Knoten s und t sowie d_s und d_t erhält und in $O(n+m)$ Zeit alle Via-Knoten ausgibt. Begründe, wieso der Algorithmus die vorgegebene Laufzeit nicht überschreitet. (3 Punkte)

Lösung: Wir nutzen eine Breitensuche von s und eine Breitensuche von t auf dem invertierten Graphen (siehe vorherige Teilaufgabe), um für jeden Knoten v sowohl $\text{dist}(s,v)$ als auch $\text{dist}(v,t)$ zu berechnen. Im Anschluss können wir alle Knoten ausgeben, für die gilt, dass $\text{dist}(s,v) = d_s$ und $\text{dist}(v,t) = d_t$.
Die lineare Laufzeit ergibt sich, da wir nach den Aufrufen der beiden Breitensuchen ($O(n+m)$) nur konstant viele Operationen für jeden Knoten durchführen müssen, um die Bedingung der Distanzen zu überprüfen.

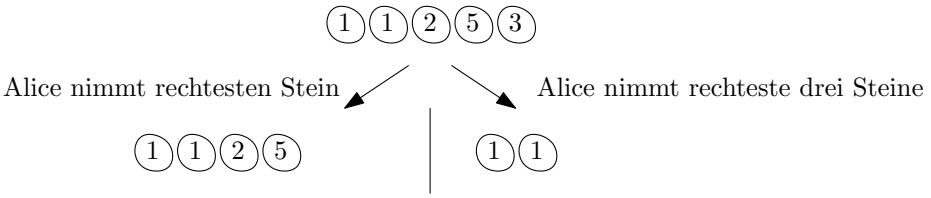
5. **Spiel mit Steinen** [10 Punkte]

Alice und Bob spielen ein Spiel. Vor ihnen liegt eine Reihe von Steinen, die mit natürlichen Zahlen beschriftet sind. Die beiden nehmen abwechselnd Steine von rechts weg. Wer am Zug ist, kann ...

- genau einen Stein (den rechtesten) nehmen, oder
- die rechtesten k Steine nehmen, wobei k die Zahl des rechtesten Steins ist. Falls weniger als k Steine übrig sind, werden alle übrigen Steine genommen.

Das Spiel endet, sobald alle Steine entfernt wurden. Wer den letzten Stein nimmt, gewinnt das Spiel.

Zum Beispiel hat Alice in der unten dargestellten Situation mit den Steinen $\langle 1, 1, 2, 5, 3\rangle$ die Wahl, einen oder drei Steine zu wählen. Nimmt sie einen Stein, hat Bob im Anschluss die Wahl, einen oder alle übrigen Steine wegzunehmen (da dann weniger als fünf Steine übrig sind). Wenn Bob alle übrigen Steine nimmt, gewinnt er. Nimmt Alice stattdessen drei Steine, kann Bob nur einen Stein wegnehmen und verliert im nächsten Zug.



Wir möchten herausfinden, ob es für Alice als Startspielerin eine *Gewinnstrategie* gibt, also ob sie so spielen kann, dass Bob nicht gewinnen kann. Im obigen Beispiel hat Alice eine Gewinnstrategie, da sie durch das Nehmen von drei Steinen sicher gewinnt.

Wir stellen die Steine als Folge natürlicher Zahlen $A = \langle a_1, \ldots, a_n\rangle$ dar.

(a) Hat Alice eine Gewinnstrategie, wenn das Spiel mit folgenden Steinen beginnt? Begründe deine Antwort. (2 Punkte)

$$A = \langle 1, 2, 3, 4, 5, 6, 7, 8, 9, 2\rangle$$

Lösung: Nein. Egal ob Alice einen oder zwei Steine entfernt, kann Bob im nächsten Zug alle verbleibenden Steine wegnehmen und dadurch gewinnen.

(b) Angenommen, Alice weiß für jedes $i < n$, ob es bei i verbleibenden Steinen für die Person, die gerade am Zug ist, eine Gewinnstrategie gibt. Wie kann Alice entscheiden, ob es bei n verbleibenden Steine eine Gewinnstrategie gibt? (3 Punkte)

Lösung: Es gibt bei n Steinen genau dann eine Gewinnstrategie, wenn in einem Zug alle Steine entfernt werden können ($a_n \geq n$) oder wenn eine Anzahl Steine entfernt werden kann, sodass es im Anschluss *keine* Gewinnstrategie gibt. Letzteres ist genau dann der Fall, wenn man bei $n - 1$ oder bei $n - a_n$ Steinen verliert.

Wir wollen nun ein dynamisches Programm entwickeln, welches berechnet, ob Alice eine Gewinnstrategie hat. Dazu legen wir ein Array X an, um die Teillösungen zu verwalten. Dabei gibt $X[i] \in \{\text{wahr}, \text{falsch}\}$ an, ob es für die Person, die aktuell am Zug ist, eine Gewinnstrategie gibt, wenn noch i Steine übrig sind.

(c) Stelle die Rekurrenz auf, mit deren Hilfe das Array X korrekt ausgefüllt werden kann. (3 Punkte)
Hinweis: Achte darauf, auch Basisfälle der Rekurrenz anzugeben, falls nötig.

Lösung:

$$X[i] = \begin{cases} \text{wahr}, & \text{falls } i \leq a_i \\ \neg X[i-1] \vee \neg X[i-a_i], & \text{sonst} \end{cases}$$

(d) Gib einen Algorithmus in Pseudocode an, der als Eingabe die Elemente $a_1, \ldots, a_n$ erhält und in $O(n)$ Zeit ausgibt, ob Alice als Startspielerin eine Gewinnstrategie hat. Verwende folgende Signatur: (2 Punkte)

`aliceCanWin`$(\langle a_1, \ldots, a_n \rangle)$: `Bool`

Lösung:

X: Array der Länge $n+1$ mit `false` initialisiert
for i from 1 to n **do**
  if $i \leq a_i$ **then**
    $X[i] \coloneqq$ `true`
  else
    $X[i] \coloneqq \neg X[i-1] \vee \neg X[i-a_i]$
return $X[n]$

6. **Customer-„Support“** [10 Punkte]

E-Corp möchte mit den langen Schlangen beim Customer-Support Geld machen und erlaubt Personen vorzudrängeln, wenn sie dafür bezahlen. Das verärgert jedoch die so übersprungenen Leute. Um zu verhindern, dass verärgerte Personen Streit anfangen, dürfen diese nie ein zweites Mal übersprungen werden. Ab und zu werden jedoch alle Verärgerten mit Schokolade besänftigt und gebeten, die Schlange zu verlassen und stattdessen eine E-Mail zu schreiben.

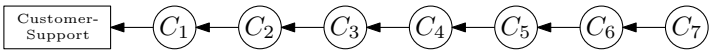
Das Unternehmen verwaltet die Warteschlange mit einer Datenstruktur, die folgende Operationen erlaubt:

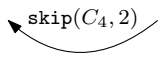
- `queue`(p): Fügt Person p am Ende der Schlange in Zeit $\Theta(1)$ ein.
- `help`(): Entfernt die vorderste Person aus der Schlange in Zeit $\Theta(1)$.
- `skip`(p, k): Die Person p wird so lange um einen Platz in der Schlange nach vorne bewegt, bis
  - insgesamt k Personen übersprungen wurden oder
  - p ganz vorne steht oder
  - direkt vor p eine verärgerte Person steht.

  Bereits verärgerte Personen dürfen also *nie* übersprungen werden. Alle übersprungenen Personen werden dann als verärgert markiert. Die Laufzeit dieser Operation ist in $\Theta(\ell)$, wobei $\ell \leq k$ die Anzahl der übersprungenen Personen ist.
- `appease`(): Entfernt alle m verärgerten Personen in Zeit $\Theta(m)$ aus der Schlange.

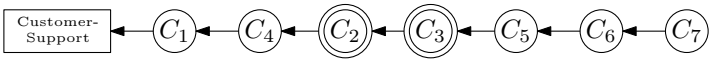
Folgende Abbildung zeigt die Schlange, nachdem $C_1, C_2, \ldots, C_7$ in dieser Reihenfolge mittels `queue` eingefügt wurden (*Zustand 1*) und einem anschließenden `skip`$(C_4, 2)$ (*Zustand 2*). Die Verärgerten C_2 und C_3 sind doppelt umrandet. Ein weiteres `skip`(C_5, k) würde für beliebiges $k > 0$ die Position von C_5 nicht verändern. Bei `skip`$(C_6, 4)$ würde C_6 nur C_5 überspringen, da C_3 bereits verärgert ist und nicht übersprungen werden kann.

Zustand 1:





Zustand 2:



(a) Zeichne den Zustand der Datenstruktur (*Zustand 3*), den man erhält, wenn man ausgehend von *Zustand 2* die Operationen `help`(), `appease()` und `skip`$(C_6, 2)$ ausführt.

Zeichne außerdem den Zustand der Datenstruktur (*Zustand 4*), den man erhält, wenn man ausgehend von *Zustand 3* die Operationen `queue`(C_8), `skip`$(C_8, 2)$ und `skip`$(C_8, 1)$ ausführt.

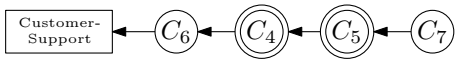
Markiere verärgerte Personen deutlich. (2 Punkte)

Zustand 3 (nach `help`(), `appease`() und `skip`(C_6, 2)):

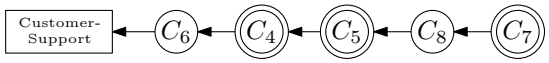
Zustand 4 (nach `queue`(C_8), `skip`(C_8, 2) und `skip`(C_8, 1)):

Lösung:

Zustand 3:



Zustand 4:



(b) Ausgehend von einer leeren Datenstruktur, gib für jedes $n \in \mathbb{N}$ eine Sequenz von Operationen an, sodass

- die Sequenz aus $\Theta(n)$ Operationen besteht,
- es eine `skip`-Operation gibt, die Laufzeit $\Theta(n)$ hat und
- es am Ende mindestens eine Person gibt, die noch $\Omega(n)$ Personen überspringen darf.

Begründe deine Antwort. (3 Punkte)

Lösung: Wir führen folgende Sequenz von Operationen aus:

1. Für alle $i \in \{1, \ldots, 2n\}$: `queue`(C_i)
2. `skip`(C_{2n}, n)

Die Sequenz besteht insgesamt aus $2n + 1 \in \Theta(n)$ Operationen. Die `skip`-Operation überspringt n nicht verärgerte Personen, benötigt also $\Theta(n)$ Zeit. Die Person C_{2n} kann noch weitere $n - 1 \in \Omega(n)$ nicht verärgerte Personen überspringen, da sie nach der ersten `skip`-Operation an Stelle n steht.

(c) Zeige, dass in jeder beliebigen Abfolge von `queue`-, `help`-, `skip`- und `appease`-Operationen jede Operation amortisiert konstante Kosten hat. (5 Punkte)

Lösung: Das kann mit Charging, Konto oder Potenzial gelöst werden.
Charging-Methode: Werden bei `skip` ℓ Personen übersprungen, teilen wir die Token von `skip` gleichmäßig auf die `queue`-Operationen der ℓ übersprungenen Personen auf. Dadurch bekommen diese `queue`-Operationen jeweils einen Token dazu und `skip` hat keine Token mehr. Auch bei `appease` werden die Kosten auf die `queue`-Operationen der entfernten Personen aufgeteilt, wodurch diese Operationen jeweils einen Token zusätzlich bekommen und `appease` keine Token mehr hat.

Es bleibt zu zeigen, dass eine `queue`-Operation nicht zu viele Token bekommen kann. Da jede Person höchstens einmal durch `skip` verärgert und höchstens einmal durch `appease` entfernt wird, bekommt jede `queue`-Operation also höchstens zwei zusätzliche Token.

Die Operationen `queue` und `help` verursachen nur konstant viele Token und bekommen keine neuen hinzu. Damit verbleiben bei jeder Operation nur konstant viele Token, womit sie also jeweils amortisiert konstante Laufzeit hat.

Konto-Methode: Jede `queue`-Operation zahlt 2 auf das Konto ein. Eine `skip`-Operation hebt ℓ ab und eine `appease`-Operation hebt m ab. Damit decken `skip` und `appease` die tatsächlichen Kosten vollständig und sind amortisiert kostenlos. Eine `queue`-Operation hat $\Theta(1)$ tatsächliche Kosten zu denen $2 \in \Theta(1)$ für die Einzahlung kommen. Eine `help`-Operation zahlt nichts ein und hebt auch nichts ab, wodurch bei ihr einfach $\Theta(1)$ tatsächliche Kosten verbleiben.

Es bleibt zu zeigen, dass das Konto nie negativ wird. Wir betrachten dazu die Anzahl nicht-verärgerter Personen n, sowie die Anzahl verärgerter Personen m, und zeigen dann induktiv, dass für den Kontostand K immer gilt $K \geq 2n + m$. Da n und m nicht negativ werden können, wird damit das gewünschte gezeigt. Zu Beginn ist $K = 0$ und es gibt keine Personen in der Schlange, weswegen die Ungleichung gilt. Wenn `appease` abhebt verringert sich die Anzahl der Verärgerten um m und der Kontostand tut das auch. Somit bleibt die Invariante erhalten. Ein `skip` verringert den Kontostand um ℓ, sodass sich der neue Konstant ergibt als $K' = K - \ell$. Nach Induktionsvoraussetzung gilt dann $K' \geq 2n + m - \ell = 2n + m - \ell + \ell - \ell = 2(n - \ell) + (m + \ell)$. Da bei einem `skip` nun ℓ der n Unverärgerten verärgert werden, erhalten wir $n' = n - \ell$ Unverärgerte und die Anzahl der Verärgerten erhöht sich auf $m' = m + \ell$. Für den neuen Kontostand gilt dann $K' \geq 2n' + m'$. Die Invariante bleibt also erhalten. Bei einem `queue` werden 2 eingezahlt. Der neue Kontostand ist also $K' = K + 2$. Nach Induktionsvoraussetzung ist das $K' \geq 2n + m + 2 = 2(n + 1) + m$. Da sich die Anzahl Unverärgerter auf $n' = n + 1$ erhöht, erhalten wir wie gewünscht $K' \geq 2n' + m$. Eine `help` Operation verringert n oder m um 1, verändert den Kontostand aber nicht, sodass die Ungleichung auch hier offensichtlich erhalten bleibt.

Potenzial-Methode: Sei n die Anzahl der Unverärgerten und m die Anzahl der Verärgerten. Wir definieren das Potenzial als $\Phi = 2n + m$. Für eine Operation betrachten wir die Änderung des Potenzials $\Delta\Phi = \Phi_{\text{neu}} - \Phi_{\text{alt}}$, wobei Φ_{alt} das Potenzial vor der Ausführung der Operation ist und Φ_{neu} das Potenzial danach. Die amortisierten Kosten ergeben sich dann aus der Summe der tatsächlichen Kosten und $\Delta\Phi$.

- `appease` hat tatsächliche Kosten m. Zudem verringert sich die Anzahl der Verärgerten von m auf 0 und damit ist die Potenzialänderung $\Delta\Phi = -m$. In Summe ergeben sich amortisierte Kosten von 0 für `appease`.
- `skip` hat tatsächliche Kosten ℓ. Von den n Unverärgerten werden ℓ verärgert. Weswegen sich n um ℓ verringert und m um ℓ erhöht. Die Potenzialänderung ist also $\Delta\Phi = -\ell$. Somit hat `skip` amortisierte Kosten von 0.
- `queue` hat tatsächliche Kosten von 1. Dabei wird n um 1 erhöht, wodurch sich das Potenzial um 2 erhöht, also $\Delta\Phi = 2$. In Summe ergeben sich also $3 \in \Theta(1)$ amortisierte Kosten für `queue`.
- `help` hat tatsächliche Kosten von 1. Dabei wird n oder m um 1 verringert, wodurch die Potenzialänderung mit $\Delta\Phi \leq -1$ abgeschätzt werden kann. In Summe bleiben also amortisierte Kosten von höchstens 0 für `help`.