Algorithmen 1 **1. Klausur** **Sommersemester 2024**

30.08.2024

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 16 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 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	

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: $\log(n) = \log_2(n)$ (3 Punkte)

$$n^2 + \log(n), \qquad (\sqrt{n}+2)(\sqrt{n}+3), \qquad \sqrt{n^5}, \qquad n^2 \log(n), \qquad \frac{n}{\log(n)}, \qquad 3^n, \qquad n^4$$

Lösung:

$$\frac{n}{\log(n)}, \qquad (\sqrt{n}+2)(\sqrt{n}+3), \qquad n^2 + \log(n), \qquad n^2 \log(n), \qquad \sqrt{n^5}, \qquad n^4, \qquad 3^n$$

(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)

$$f(n) = \sum_{i=0}^{\log_2(n)} \frac{2^i}{n}$$

Lösung: $f(n) \in \Theta(1)$

(c) Bestimme für folgende Situationen, welche möglichst einfache und effiziente Datenstruktur jeweils geeignet ist. Gib außerdem an, wie die drei geforderten Operationen und Anfragen i), ii) und iii) 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) $2\times$ Element vorne löschen
- ii) Element hinten einfügen
- iii) durch Liste iterieren und alle Elemente ausgeben

1. Bei einem Marathon sind viele Läuferinnen dabei.
   - i) Die Läuferinnen starten nacheinander in einer bestimmten Reihenfolge.
   - ii) Die Läuferin auf Rang i überholt die Läuferin direkt vor ihr.
   - iii) Wir möchten regelmäßig die Reihenfolge aller Läuferinnen anfragen.

   Lösung: Datenstruktur: dynamisches Array
   - i) Array initialisieren/Elemente hinten einfügen
   - ii) Tausche Element i und $i-1$
   - iii) Alle Elemente von vorne bis hinten ausgeben

2. Am KIT gibt es viele Lerngruppen.
   - i) Zwei Lerngruppen tun sich zusammen, um für eine Klausur zu lernen.
   - ii) Neue Studis wechseln an das KIT und sind initial alleine in einer Lerngruppe.
   - iii) Wir möchten wissen, ob zwei bestimmte Studierende i und j in der selben Lerngruppe sind.

Lösung: Datenstruktur: Union Find

i) union mit zwei Lerngruppen

ii) Füge neue Partition hinzu

iii) Prüfe, ob i und j in gleicher Partition

(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) die Worst-Case-Laufzeit benötigt. Gib außerdem die Worst-Case-Laufzeit im O-Kalkül an. (2 Punkte)

```
Function magic(Array A):
    for i ∈ {0, ..., n − 1} do
        for j ∈ {0, ..., n − 1} do
            if A[i] ≠ A[j] then
                return False
    return True
```

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

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

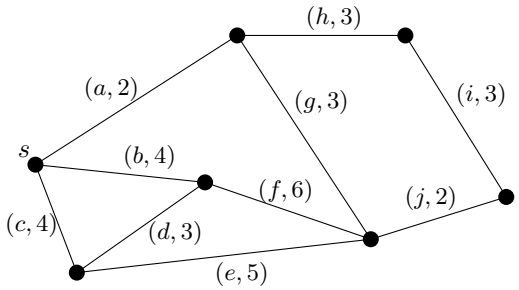
Worst-Case-Laufzeit:

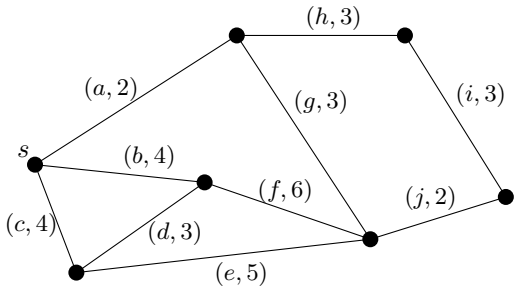
Lösung: $O(n^2)$

2. **Viele MSTs** [10 Punkte]

In der Vorlesung haben wir für die Bestimmung von Minimum Spanning Trees angenommen, dass alle Kantengewichte unterschiedlich sind. Nun möchten wir zulassen, dass verschiedene Kanten das gleiche Gewicht haben.

(a) Im folgenden Graphen hat jede Kante ein eindeutiges alphabetisches Label und ein Gewicht. Führe Prims Algorithmus darauf aus und kennzeichne die Kanten des MSTs, den Prim ausgibt. Beginne beim linken Knoten s. Wenn Prim verschiedene Kanten mit gleichem Kantengewicht zur Auswahl hat, wähle zuerst die alphabetisch *kleinere* Kante. Gib außerdem die Reihenfolge an, in der die Kanten zum MST hinzugefügt werden. (3 Punkte)

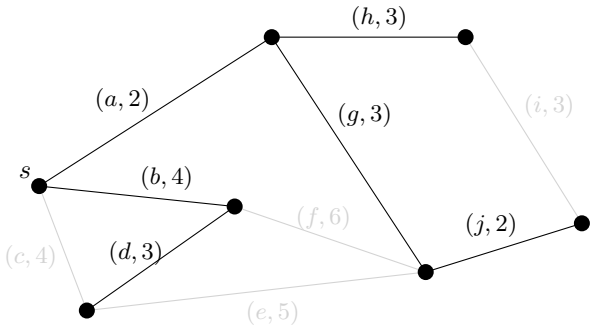




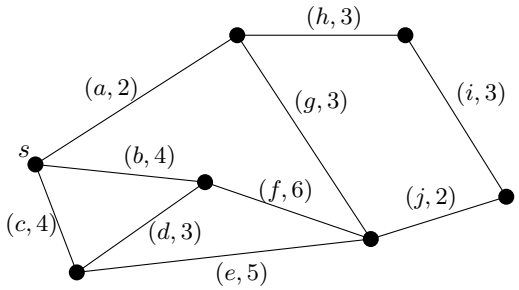
Kopie desselben Graphen, falls du dich beim ersten vertan hast. Markiere deutlich, welche Lösung zu bewerten ist.

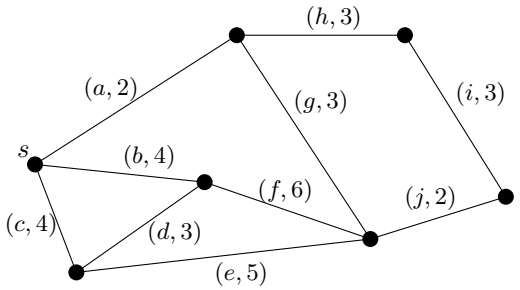
Reihenfolge, in der die Kanten zum MST hinzugefügt werden:

Lösung: a, g, j, h, b, d



(b) Führe Prims Algorithmus erneut auf dem folgenden Graphen aus. Wähle aber nun die alphabetisch *größere* Kante aus, wenn Prim verschiedene Kanten mit gleichem Kantengewicht zur Auswahl hat. Kennzeichne die Kanten des MSTs, den Prim ausgibt. Beginne beim linken Knoten s. Gib außerdem die Reihenfolge an, in der die Kanten zum MST hinzugefügt werden. (3 Punkte)

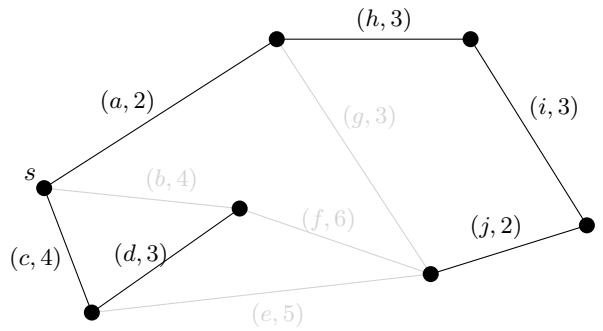




Kopie desselben Graphen, falls du dich beim ersten vertan hast. Markiere deutlich, welche Lösung zu bewerten ist.

Reihenfolge, in der die Kanten zum MST hinzugefügt werden:

Lösung: a, h, i, j, c, d



(c) Wie viele verschiedene minimale Spannbäume hat der obige Graph? Begründe deine Antwort kurz.(1 Punkt)

Lösung: 6. In jedem MST ist genau eine Kante aus $\{b, c\}$ enthalten und es sind genau zwei Kanten aus $\{g, h, i\}$. Das macht $2 \cdot 3 = 6$ Möglichkeiten.

(d) Wir betrachten im Folgenden nur zusammenhängende Graphen. Zeige oder widerlege die folgenden Aussagen. (3 Punkte)

1. Wenn G zwei unterschiedliche Kanten mit gleichem Kantengewicht enthält, dann ist der MST von G nicht eindeutig.

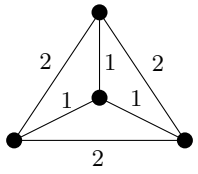
   Lösung: Die Aussage ist falsch.

   Der MST ist eindeutig (der Graph selbst), obwohl zwei Kanten das gleiche Gewicht haben.

2. Wenn G einen Kreis enthält, auf dem alle Kanten das gleiche Kantengewicht haben, dann ist der MST von G nicht eindeutig.

   Lösung: Die Aussage ist falsch.

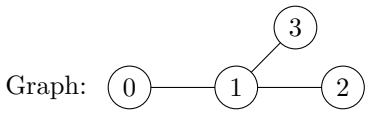
   Der MST ist eindeutig (die Kanten mit Gewicht 1), obwohl es einen Kreis gibt, auf dem alle Kanten Gewicht 2 haben.

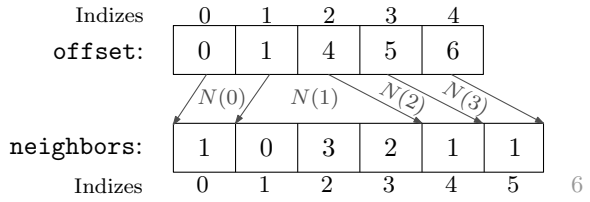
3. **Adjazenzspeicher** [10 Punkte]

In dieser Aufgabe betrachten wir *Adjazenzspeicher*, eine neue Datenstruktur zum Speichern von Graphen. Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit n Knoten und m Kanten. Die Knoten nummerieren wir von 0 bis $n - 1$. Ein Adjazenzspeicher für G besteht aus einem Offset-Array `offset` der Länge $n + 1$ und einem Nachbarschaft-Array `neighbors` der Länge $2m$.

Im Nachbarschafts-Array stehen nacheinander zuerst alle Nachbarn von Knoten 0, dann alle Nachbarn von Knoten 1 und so weiter. Das Offset-Array gibt die Grenzen im Nachbarschafts-Array an. Für einen Knoten i speichern wir also die Nachbarn an den Stellen mit Index `offset`$[i]$ bis einschließlich `offset`$[i + 1] - 1$ in `neighbors`.

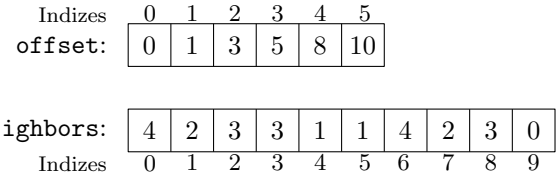
Beispiel:



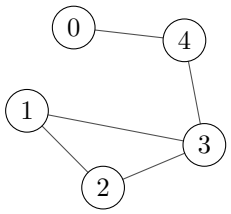


6

(a) Zeichne den Graphen, der von dem folgenden Adjazenzspeicher repräsentiert wird. Beschrifte die Knoten mit ihrem Index im Adjazenzspeicher. (3 Punkte)



Lösung:



(b) Gegeben sei ein Graph G als Adjazenzspeicher. Beschreibe einen Algorithmus in Worten, der für G entscheidet, ob zwei gegebene Knoten i und j adjazent sind. Gib außerdem die Laufzeit des Algorithmus in Abhängigkeit von den Knotengraden der gegebenen Knoten an. (2 Punkte)

Lösung: Die Nachbarschaft von Knoten i steht im Nachbarschaftsarray zwischen den Indizes offset$[i]$ und offset$[i + 1] - 1$. Wir prüfen, ob j in diesem Intervall im Nachbarschaftsarray enthalten ist. Die Größe des Intervalls entspricht genau dem Grad von i. Damit ist die Laufzeit $O(\deg(i))$.

Wenn wir in der Nachbarschaft des Knoten mit kleinerem Grad suchen, erhalten wir eine Laufzeit von $O(\min\{\deg(i), \deg(j)\})$.

(c) Du darfst nun einen *sortierten* Adjazenzspeicher annehmen. Das heißt, dass die Nachbarschaft jedes Knoten im Nachbarschaftsarray (also `neighbors`$[$`offset`$[i], \ldots,$ `offset`$[i+1]-1]$ für Knoten i) aufsteigend sortiert ist. Beschreibe einen Algorithmus in Worten, der für einen als sortierten Adjazenzspeicher gegebenen Graphen entscheidet, ob zwei gegebene Knoten i und j adjazent sind. Gib außerdem die Laufzeit des Algorithmus in Abhängigkeit von den Knotengraden der gegebenen Knoten an. (2 Punkte)

Lösung: In einem sortierten Adjazenzspeicher können wir mit einer binären Suche in $O(\log(\deg(i)))$ entscheiden, ob j in der Nachbarschaft von i enthalten ist.

Wenn wir in der Nachbarschaft des Knoten mit kleinerem Grad suchen, erhalten wir eine Laufzeit von $O(\min\{\log(\deg(i)), \log(\deg(j))\})$.

(d) Wir möchten einen als Adjazenzliste gegebenen ungerichteten Graphen G zu einem (nicht sortierten) Adjazenzspeicher konvertieren. Der Graph G hat n Knoten und m Kanten, wobei die Knoten von 0 bis $n-1$ nummeriert sind. Vervollständige dafür den folgenden Pseudocode. Der Algorithmus soll eine Laufzeit von $O(n+m)$ haben. (3 Punkte)

Function `konvertiere`(*Adjazenzliste A*)**:**

```
offset : Array = ⟨⟩                    // leeres Array
neighbors : Array = ⟨⟩                 // leeres Array
```

Lösung:

```
offset.pushBack(0)
for i ∈ {0, ..., n − 1} do
    for j ∈ A[i] do
        neighbors.pushBack(j)
    offset.pushBack(offset[i] + A[i].size())
```

return offset, neighbors

4. **Hashing** [10 Punkte]

In dieser Aufgabe sei U eine Teilmenge der natürlichen Zahlen.

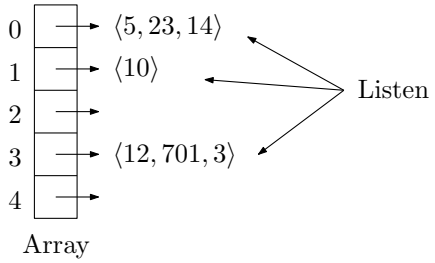
Betrachte die folgende Hashfunktion $h_{\text{sum}} : U \to [0, m)$ mit $h_{\text{sum}}(x) = \text{sum}(x) \bmod m$, wobei $\text{sum}(x)$ der Summe aller Ziffern von x entspricht. Zum Beispiel ist $\text{sum}(1231) = 1 + 2 + 3 + 1 = 7$.

(a) Sei $m = 5$. Die folgenden Zahlen sollen nacheinander in eine Hashtabelle der Größe 5 mit der Hashfunktion h_{sum} eingefügt werden:

5, 23, 12, 701, 14, 3, 10

Gib den vollständigen Zustand der Hashtabelle nach Einfügen aller obigen Zahlen an. Gib außerdem deutlich an, wo die Hashtabelle welche anderen Datenstrukturen verwendet. (2 Punkte)

Lösung:



(b) Gib für jedes $m \in \mathbb{N}$ eine Menge von m paarweise unterschiedlichen Zahlen an, (2 Punkte)

1. für die h_{sum} eine möglichst *gute* Hashfunktion ist.

   Lösung: $1, 11, 111, 1111, \ldots, 1^m$ (Zahlen mit $i \in \{1, \ldots, m\}$ Einsen). Die Quersumme der Zahlen ist dann $1, 2, \ldots, m$. Weil die Hashtabelle m Buckets hat, wird jeder Bucket genau einmal belegt.

2. für die h_{sum} eine möglichst *schlechte* Hashfunktion ist.

   Lösung: $10, 100, 1000, 10000, \ldots, 10^m$ (eine Eins mit $i \in \{1, \ldots, m\}$ Nullen). Jede Zahl hat eine Quersumme von 1. Damit landen alle Elemente in dem selben Bucket.

(c) Sei nun $U = \{0, 1, 2, 3, 4\}$ und $m = 3$. Gib eine Familie $\mathcal{H} = \{h_1, h_2\}$ aus zwei Hashfunktionen $h_1, h_2 \colon U \to \{0, 1, 2\}$ an, sodass $\mathcal{H}$ die folgende Eigenschaft hat:

Für alle $a, b \in U$ mit $a \neq b$ und einer zufällig gezogenen Hashfunktion $h \in \mathcal{H}$ gilt $Pr[h(a) = h(b)] \leq \frac{1}{2}$.

Fülle dazu die beiden folgenden Tabellen aus, um h_1 und h_2 anzugeben. Begründe kurz, warum deine Lösung die geforderte Eigenschaft erfüllt. (3 Punkte)

x	0	1	2	3	4
$h_1(x)$					

x	0	1	2	3	4
$h_2(x)$					

Lösung:

x	0	1	2	3	4
$h_1(x)$	0	0	1	1	2

x	0	1	2	3	4
$h_2(x)$	0	1	0	1	2

Jedes Paar kollidiert für höchstens eine Hashfunktion. Da es nur zwei Hashfunktionen gibt, ist die Kollisionswahrscheinlichkeit für jedes Paar $\leq \frac{1}{2}$.

Zur Erinnerung: Sei $\mathcal{H}$ eine Familie von Hashfunktionen mit $h : U \to [0, m)$ für alle $h \in \mathcal{H}$. Dann ist $\mathcal{H}$ *universell*, wenn für alle $a, b \in U$ mit $a \neq b$ und einer zufällig gezogenen Funktion $h \in \mathcal{H}$ gilt, dass

$$\Pr[h(a) = h(b)] \leq \frac{1}{m}.$$

(d) Sei $|U| > m$. Widerlege folgende Aussage: Es gibt eine universelle Familie $\mathcal{H}$ von Hashfunktionen mit $|\mathcal{H}| < m$. (3 Punkte)

Lösung: Diese Aussage ist falsch. Sei $h \in \mathcal{H}$ eine beliebige Hashfunktion. Da $|U| > m$ ist, gibt es mindestens zwei Elemente $a, b \in U$ mit $h(a) = h(b)$.
Für h gibt es bei a, b also mindestens eine Kollision. Da es nur wenige Hashfunktionen gibt, ist die Wahrscheinlichkeit für eine Kollision schon zu hoch. Die Wahrscheinlichkeit für eine Kollision ist:

$$\Pr[h(a) = h(b)] = \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \Pr[h(a) = h(b)] = \frac{1}{|\mathcal{H}|} \left(1 + \sum_{h \in \mathcal{H} \setminus \{h\}} \Pr[h(a) = h(b)] \right) \geq \frac{1}{|\mathcal{H}|} > \frac{1}{m}.$$

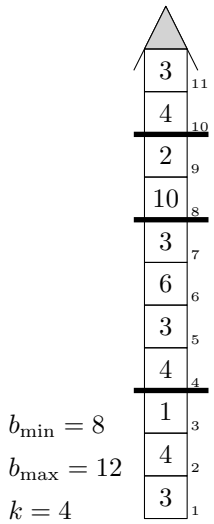
5. **Etagenzuweisung** [10 Punkte]

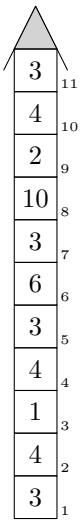
Wir wollen leerstehende Büros eines Hochhauses vermieten. Unser Hochhaus hat $n \in \mathbb{N}$ Stockwerke, wobei in Stockwerk i $a_i \in \mathbb{N}$ Büros zur Verfügung stehen. Insgesamt werden $k \in \mathbb{N}$ verschiedene Firmen Büros mieten. Jeder Firma wollen wir ein zusammenhängendes Intervall an Stockwerken zuweisen, sodass sich die Stockwerke von verschiedenen Firmen nicht überschneiden. Außerdem müssen alle Stockwerke vergeben werden.

Jede Firma soll ungefähr gleich viele Büros bekommen. Dafür führen wir eine untere und obere Schranke ein, die angibt, wie viele Büros jede Firma bekommen soll. Diese Schranken sind für alle Firmen gleich, jede Firma soll also mindestens $b_{\min}$ Büros und höchstens $b_{\max}$ Büros bekommen.

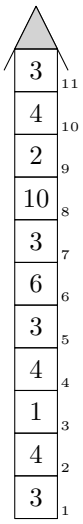
Das Problem, für gegebene Anzahlen $(a_1, \ldots, a_n)$ von Büros pro Stockwerk und Schranken $b_{\min}$ und $b_{\max}$ eine gültige Zuweisung von Etagen an Firmen zu berechnen, nennen wir ETAGENZUWEISUNG.

Wir betrachten das folgende Beispiel eines Hochhauses mit $n = 11$ Stockwerken, $b_{\min} = 8$, $b_{\max} = 12$ und $k = 4$ Firmen. Die eingezeichnete Aufteilung ist nicht gültig, da z.B. die Firma mit den Stockwerken 4 bis 7 insgesamt 16 Büros bekommt, was zu viel ist.





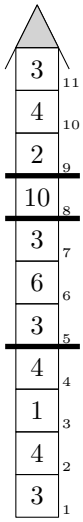
Deine Lösung



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

(a) Zeichne in dem obigen Beispiel eine gültige Aufteilung der Büros auf die Firmen ein oder gib die Intervalle von zugewiesenen Stockwerke an. (2 Punkte)

Lösung:



(b) Angenommen für alle $i < n$ ist bereits bekannt, ob die Stockwerke $\{1, \ldots, i\}$ an $k-1$ Firmen vergeben werden können. Beschreibe, wie entschieden werden kann, ob n Stockwerke an k Firmen vergeben werden können. (2 Punkte)

Lösung: Wir betrachten alle möglichen Intervalle $[x, n]$, die an die oberste Firma vergeben werden können. Hierfür muss die Summe der Anzahl der Büros in diesem Intervall zwischen $b_{\min}$ und $b_{\max}$ liegen. Außerdem müssen die ersten $x-1$ Stockwerke an $k-1$ Firmen vergeben werden können. Es ist genau dann möglich die Stockwerke $\{1, \ldots, n\}$ an k Firmen zu vergeben, wenn es ein solches x gibt.

Wir wollen nun ein dynamisches Programm für das Problem ETAGENZUWEISUNG formulieren. Dabei genügt es uns zu entscheiden, ob es eine gültige Zuweisung gibt. Hierfür wollen wir für jedes $i \in \{0, \ldots, n\}$ und jedes $j \in \{0, \ldots, k\}$ eine Teillösung $X[i][j]$ in einem Array X speichern.

(c) Gib an, welche Bedeutung eine Teillösung $X[i][j]$ 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: Die Teillösung $X[i][j]$ beschreibt, ob es möglich ist, die ersten i Stockwerke an genau j Firmen zu vermieten, wobei jede Firma zwischen $b_{\min}$ und $b_{\max}$ viele Büros erhält. Bei $j = 0$ ist der Wert immer `false`, außer wenn $i = 0$. Also

$$X[0][0] = \texttt{true}, \text{ sowie } X[i][0] = \texttt{false} \text{ für } i > 0.$$

Um die Rekurrenz aufzustellen definieren wir zunächst eine Hilfsmenge

$$S_i = \left\{ p \in \{0, \ldots, i\} \mid b_{\min} \leq \sum_{q=p+1}^{i} a_q \leq b_{\max} \right\}.$$

Für alle Indizes $p \in S_i$ gilt, dass das Intervall $[p+1, i]$ einer Firma zugewiesen werden kann.
Im Fall $j > 0$ überprüfen wir, ob es eine Zahl p gibt, sodass die oberste Firma die Büros zwischen $p+1$ und i bekommen darf und die restlichen Stockwerke an die anderen $j-1$ Firmen verteilt werden können. Damit lautet die Rekurrenzgleichung:

$$X[i][j] = \bigvee_{p \in S_i} X[p][j-1]$$

(d) Was ist die Laufzeit, um alle Teillösungen mittels eines dynamischen Programms zu berechnen? Begründe deine Antwort. (1 Punkt)

Lösung:
Eine Teillösung $X[i][j]$ auszurechnen benötigt $O(n)$ Zeit. Da es insgesamt $O(n \cdot k)$ Teillösungen gibt, kann das Problem in $O(n^2 \cdot k)$ Zeit gelöst werden.

6. **Paketmanager** [10 Punkte]

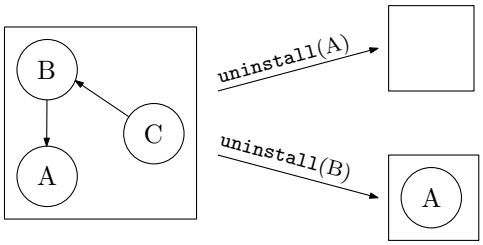
Wir möchten die installierten Pakete eines Betriebssystems verwalten. Dabei können Pakete installiert und deinstalliert werden. Manche Pakete benötigen andere Pakete und funktionieren nur, wenn alle benötigten Pakete auch installiert sind.

Wir repräsentieren den Paketmanager als gerichteten Graphen, wobei die Pakete die Knoten sind. Eine Kante von Paket A zu Paket B bedeutet, dass Paket A Paket B benötigt. Wir sagen außerdem, dass Paket A das Paket B *transitiv benötigt*, wenn es einen gerichteten Pfad von A nach B gibt.

Um Pakete zu verwalten, werden folgende Operationen bereitgestellt.

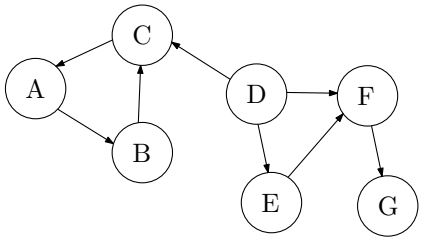
- `install(A : Paket)` – Fügt einen neuen Knoten hinzu, installiert also ein neues Paket. Laufzeit $\Theta(1)$.
- `addUsage(user : Paket, needed : Paket)` – Fügt eine gerichtete Kante vom Paket `user` zum Paket `needed` ein, falls diese noch nicht existiert. Das heißt, Paket `needed` wird von Paket `user` benötigt. Laufzeit $\Theta(1)$.
- `uninstall(A : Paket)` – Löscht Paket A und alle Pakete, die A transitiv benötigen. Löscht außerdem die Kanten, bei denen ein inzidenter Knoten gelöscht wurde. Laufzeit $\Theta(x + y)$, wobei x die Anzahl gelöschter Knoten und y die Anzahl gelöschter Kanten ist.

Beispiel: Wir fügen in eine leere Datenstruktur mit `install`(A), `install`(B) und `install`(C) drei neue Pakete hinzu. Außerdem fügen wir mit `addUsage`(B, A) und `addUsage`(C, B) zwei gerichtete Kanten hinzu.



Wird nun `uninstall`(A) ausgeführt, werden alle Knoten und Kanten gelöscht, da sowohl B als auch C Paket A transitiv benötigen. Wird *stattdessen* `uninstall`(B) ausgeführt, bleibt Paket A übrig, da dieses Paket B nicht transitiv benötigt.

(a) Betrachte den folgenden Zustand des Paketmanagers. Zeichne den Zustand, nachdem die beiden Operationen `uninstall`(E) und `uninstall`(B) ausgeführt wurden. (1 Punkt)



Zustand 1

Lösung: Es bleiben nur die Knoten F und G und die Kante (G, F) übrig.

(b) Betrachte Zustand 1 aus der vorherigen Teilaufgabe. Gib eine Folge mit möglichst wenigen `uninstall`-Operationen an, sodass alle Pakete gelöscht werden. (1 Punkt)

Lösung: `uninstall`(A), `uninstall`(G)

(c) Ausgehend von einer leeren Datenstruktur, gib für jedes $n \in \mathbb{N}$ eine Sequenz von $O(n)$ Operationen an, die $O(n)$ `install`-Operationen und $O(n)$ `addUsage`-Operationen enthält, sowie eine `uninstall`-Operation mit $\Theta(n)$ Kosten. Begründe deine Antwort kurz. (3 Punkte)

Lösung: Konstruiere Pfad: Installiere n Pakete $\{1, \ldots, n\}$ und füge Kanten $(i, i+1)$ für alle $i \in \{1, \ldots n-1\}$ hinzu. `uninstall`(n) löscht die Pakete $1, \ldots, n$, benötigt also $\Theta(n)$ Zeit.

(d) Zeige, dass in jeder beliebigen Abfolge von `install`-, `addUsage`- und `uninstall`-Operationen jede Operation amortisiert konstante Kosten hat. (5 Punkte)

Lösung:

Aggregation

Betrachte eine beliebige Folge von n `install`-, m `addUsage`- und k `uninstall`-Operationen. Über die ganze Folge hinweg werden n Knoten eingefügt und bis zu n Knoten wieder gelöscht. Genauso, werden m Kanten eingefügt und bis zu m Kanten gelöscht. Damit entstehen Gesamtkosten von höchstens $2(n+m)$. Teilen wir die Gesamtkosten durch die Anzahl Operationen erhalten amortisierte Kosten pro Operation von

$$\frac{2(n+m)}{n+m+k} \leq \frac{2(n+m)}{n+m} = 2 \in \Theta(1).$$

Charging

Die Operationen `install` und `addUsage` haben jeweils nur konstante Kosten und müssen daher keine Tokens verteilen. Die Operation `uninstall` erzeugt $\Theta(x+y)$ viele Kosten und löscht x viele Knoten und y viele Kanten. Es werden x Kostentoken auf diejenigen `install`-Operationen verteilt, welche die Knoten erzeugt haben und y Token auf die `addUsage`-Operationen, welche die Kanten erzeugt haben. Da Knoten und Kanten nur einmal gelöscht werden, erhält jede Operation maximal 1 zusätzliches Token. Es können also alle Tokens von `uninstall` verteilt werden und die amortisierten Kosten sind in $O(1)$.

Kontomethode

Die `install`- und `addUsage`-Operationen haben jeweils Kosten 1 und zahlen zusätzlich 1 Guthaben in das Konto ein. Eine `uninstall`-Operation mit Kosten $x+y$ hebt $x+y$ vom Konto ab und deckt damit die Kosten vollständig. Damit erhält man amortisiert konstante Kosten pro Operation, wenn wir zeigen, dass das Konto nie negativ wird. Man kann leicht die Invariante einsehen, dass der Kontostand zu jedem Zeitpunkt $n+m \geq 0$ ist, wobei n die Anzahl Knoten und m die Anzahl Kanten ist. Zu Beginn gilt die Invariante, da der Graph leer und der Kontostand 0 ist. Jede Operation verändert den Kontostand jetzt auf genau die Art, auf die sich auch die Größe des Graphen $(n+m)$ ändert. Damit bleibt die Invariante erhalten.

Potentialmethode

Wir definieren folgendes Potential $\Phi = n + m \geq 0$, wobei n die aktuelle Anzahl der Knoten und m die aktuelle Anzahl der Kanten ist. Dann gilt für die amortisierten Kosten:

- `install` hat tatsächliche Kosten $\Theta(1)$. Da ein Knoten hinzugefügt wird, ist $\Phi_{\text{neu}} - \Phi_{\text{alt}} = 1$. Insgesamt hat `install` amortisierte Kosten $1 + 1 \in \Theta(1)$.
- `addUsage` funktioniert analog zu `install`, hat also auch amortisiert konstante Kosten.
- `uninstall` hat tatsächliche Kosten $\Theta(x+y)$, wobei x die Anzahl der gelöschten Knoten und y die Anzahl der gelöschten Kanten ist. Da $x+y$ Knoten und Kanten gelöscht werden, gilt $\Phi_{\text{neu}} - \Phi_{\text{alt}} = -x-y$. Weil $x+y-x-y=0$, hat auch `uninstall` amortisierte konstante Kosten.