Albert-Ludwigs-Universität
Institut für Informatik
Prof. Dr. F. Kuhn

# Klausur Informatik 2

Mittwoch, 8. März 2019, 13:00-16:00

Name: .....

Matrikel Nr.: .....

Unterschrift: .....

## Erst öffnen wenn die Klausuraufsicht die Erlaubnis erteilt!

- Legen Sie Ihren **Studentenausweis** sichtbar vor sich.
- Schreiben Sie Ihren **Namen** und Ihre **Matrikelnummer** an die vorgesehenen Stellen.
- **Unterschreiben Sie** diese Seite um zu bestätigen, dass Sie die Fragen ohne unerlaubte Hilfsmittel beantwortet haben und die Klausuraufsicht über (gesundheitliche) Probleme informiert haben.
- Dies ist eine **Open Book** Klausur weshalb alle gedruckten und handgeschriebenen Materialien erlaubt sind. **Elektronische Hilfsmittel** sind **nicht** erlaubt.
- Schreiben Sie **lesbar** und nur mit **dokumentenechten** Stiften. Benutzen Sie **keine rote** Farbe und **keinen Bleistift**!
- Es wird **nur eine Lösung pro Aufgabe** gewertet. Vergewissern Sie sich, dass Sie zusätzliche Lösungen durchstreichen, andernfalls wird die schlechteste Lösung gewertet.
- Detaillierte Schritte können Ihnen zu **Teilpunkten** verhelfen falls das Endergebnis falsch ist.
- Die Schlüsselwörter **Zeigen Sie...**, **Beweisen Sie...**, **Begründen Sie...** oder **Leiten Sie ... her** zeigen an, dass Sie Ihre Antwort sorgfältig und gegebenenfalls formal begründen müssen.
- Die Schlüsselwörter **Geben Sie ... an** zeigen an, dass Sie lediglich die geforderte Antwort und keine Begründung liefern müssen.
- Die folgenden Regeln gelten **überall**, außer sie werden explizit außer Kraft gesetzt.
- Bei Laufzeitfragen ist nur die **asymptotische Laufzeit** notwendig.
- Wenn Sie einen Algorithmus angeben sollen, so können Sie Pseudocode angeben. Eine **ausreichend detaillierte** Beschreibung der Funktionsweise Ihres Algorithmus genügt jedoch.
- Algorithmen aus der Vorlesung **können als Blackbox** verwendet werden.
- **Lesen Sie jede Aufgabe sorgfältig** durch und stellen Sie sicher dass Sie diese verstanden haben!
- Falls Sie eine Frage zur Aufgabenstellung haben, geben Sie der Klausuraufsicht ein **Handzeichen**.
- Schreiben Sie Ihren Namen auf **alle Blätter**!

| Aufgabe | 1 | 2 | 3 | 4 | 5 | 6 | 7 | **Total** |
|---|---|---|---|---|---|---|---|---|
| Maximum | 23 | 9 | 15 | 7 | 18 | 21 | 27 | 120 |
| Punkte | | | | | | | | |

# Aufgabe 1: Kurze Fragen *(23 Punkte)*

(a) Sortieren Sie die folgenden Funktionen **aufsteigend** nach ihrem asymptotischen Wachstum.

| | | | |
|---|---|---|---|
| $n^n$ | $\log(n^n)$ | $2^{\log n}$ | $(\log n)^n$ |
| $n \log n$ | $3^n$ | $\log\left((\sqrt{n})^n\right)$ | $\left((2n)!\right)^2$ |

*Hinweise:* $\log(\cdot)$ *bezeichnet den Logarithmus zur Basis* 2. *Zwischen jeweils aufeinanderfolgenden Funktionen* $f(n), g(n)$ *in Ihrer Sortierung notieren Sie "*$f(n) < g(n)$*" falls* $f(n) \in \mathcal{O}\big(g(n)\big)$ *und* $g(n) \notin \mathcal{O}\big(f(n)\big)$ *gilt. Notieren Sie "*$f(n) = g(n)$*" wenn* $f(n) \in \mathcal{O}\big(g(n)\big)$ *und* $g(n) \in \mathcal{O}\big(f(n)\big)$ *gilt.* *(4 Punkte)*

(b) Gegeben sei ein gerichteter Graph $G = (V, E)$, abgespeichert via **Adjazenzlisten**, und ein Knoten $v \in V$. Geben Sie einen Algorithmus an, mit welchem Sie in möglichst kurzer Zeit die Menge $U = \{u \in V : \exists \text{ Pfad von } u \text{ nach } v\}$ finden, d.h., alle Knoten $u$ von welchen aus ein Pfad nach $v$ existiert. *(4 Punkte)*

(c) Gegeben seien $k$ **bereits sortierte** Arrays $A_1, \ldots, A_k$ mit je $m$ Sortierschlüsseln. Beschreiben Sie einen Algorithmus der ein **sortiertes** Array $A$ der Größe $km$ aus allen Werten der Arrays $A_1, \ldots, A_k$ berechnet und dabei $\mathcal{O}(km \log k)$ Vergleiche benötigt. *(4 Punkte)*

(d) Sei $A$ ein **bereits sortiertes** Array mit $n$ paarweise verschiedenen Sortierschlüsseln. Es ist möglich **zwei** Sortierschlüssel in $A$ zu vertauschen, um ein Array $A'$ mit folgender Eigenschaft zu erhalten: Um das Array $A'$ korrekt zu sortieren werden $\Omega(n)$ Durchläufe der äußeren Schleife des Sortieralgorithmus **Bubblesort** (wie in den Vorlesungsunterlagen) benötigt. Geben Sie die Vertauschung an und begründen Sie Ihre Antwort. *(5 Punkte)*

(e) Gegeben seien die Hashfunktionen $h_1(x) = x \mod m$ und $h_2(x) = 1 + (x \mod (m-1))$. Fügen Sie die Schlüssel 28, 59, 47, 13, 39 nacheinander in die unten stehende Hashtabelle der Größe $m = 11$ ein, indem Sie die Methode des **Doppel-Hashing** nutzen. *(4 Punkte)*

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

(f) Nennen Sie einen Vorteil des **Doppel-Hashing** gegenüber Hashing mit **quadratischer Sondierung**. *(2 Punkte)*

# Musterlösung

(a) *Korrekturhinweis: Einen Punkt Abzug für jedes aufeinanderfolgende Paar mit falscher Reihenfolge bzw. falschem Zeichen* $<$ *bzw.* $=$.

$$\begin{array}{ccccccccc} & 2^{\log n} & < & \log(n^n) & = & \log\left((\sqrt{n})^n\right) & = & n\log n & \\ < & 3^n & < & (\log n)^n & < & n^n & < & \left((2n)!\right)^2 \end{array}$$

(b) Alle Knoten zu finden, von denen aus ein Pfad nach $v$ existiert, ist fast das Gleiche wie alle Knoten zu finden, die von $v$ aus erreicht werden können – nur umgekehrt. Betrachte den Graphen $G' = (V, E')$, in welchem jede Kante aus $E$ umgedreht existiert, d.h., $E' = \{(y, x) : (x, y) \in E\}$. Wenn von $u$ ein Pfad nach $v$ in $G$ existiert, dann existiert ein Pfad von $v$ nach $u$ in $G'$. D.h., wir führen BFS auf $G'$ aus, startend mit $v$. DFS geht genauso. Alle Knoten, die von $v$ aus in $G'$ erreicht werden können, fügen wir zu $U$ hinzu. *(2 Punkte)*

*Korrekturhinweis: Die zwei Punkte gibt es für den Hinweis dass man DFS auf dem "umgedrehten" Graphen $G'$ ausführt. Einfach nur DFS auf $G$ auszuführen gibt 1 Punkt da immerhin auf ungerichteten Graphen korrekt (laut Aufgabenstellung ist der Graph aber gerichtet).*

Das Einzige, was wir noch sicherstellen müssen, ist, dass wir $G'$ schnell aus $G$ kreieren können. Dazu gehen wir ähnlich vor wie in der vorhergehenden Aufgabe, indem wir erst eine Kopie des Knoten-Arrays erstellen und dann durch die Adjazenzlisten durchgehen. Gibt es in der Adjazenzliste von $x$ einen Zeiger zu $y$, d.h., eine Kante $(x, y)$ im Graphen, dann machen wir im zweiten Array einen Eintrag in der Adjazenzliste von $y$, mit einem Zeiger auf $x$. Jede Kante wird so einmal kopiert und dabei umgedreht. *(2 Punkte)*

Die Laufzeit von beiden Algorithmen ist in der Größenordnung des Speichplatzbedarfs von Adjazenzlisten, d.h., $O(n + m)$, und das ist optimal, da wir jede Kante und jeden Knoten mindestens einmal anschauen müssen.

*Korrekturhinweis: Eine Laufzeitanalyse muss laut Aufgabenstellung nicht erbracht werden. Algorithmen mit schlechteren Laufzeiten die aber immerhin polynomiell in der Eingabegröße sind geben bis zu einem Punkt Abzug.*

(c) Beobachtung: Wir haben eine Situation ähnlich zum Combine-Schritt bei Mergesort, bei der $k$ bereits sortierte Teilarrays zusammengeführt werden müssen. Sei $k_1 := k$ und seien die Arrays $A_1^1 := A_1, \ldots, A_{k_1}^1 := A_k$ die inital gegebenen, bereits sortierten Arrays.

Wir benutzen den in der Vorlesung beschriebenen Merge-Schritt iterativ als Black-box um sortierte Arrays $A_1^i, \ldots, A_{k_i}^i$ jeweils paarweise in sortierte Arrays $A_1^{i+1}, \ldots, A_{k_{i+1}}^{i+1}$ zusammenzuführen, solange bis $k_i = 1$ ist. Falls $k_i > 1$ und ungerade ist kombinieren wir nur die ersten $(k_i - 1)/2$ Paare.

Seien $A_1^{i+1}, \ldots, A_{k_{i+1}}^{i+1}$ die Arrays nach dem $(i+1)$-ten Merge Schritt. Es gilt dass $k_{i+1} = \lceil k_i/2 \rceil$. Nach spätestens zwei Merge-Schritten gilt

$$k_{i+2} \leq \lceil k_{i+1}/2 \rceil \leq \lceil \lceil k_i/2 \rceil /2 \rceil = \lceil k_i/4 \rceil \overset{k_i \geq 2}{\leq} k_i/2,$$

d.h. nach spätestens zwei Merge-Schritten halbiert sich die restliche Anzahl der Arrays. Also ist das Verfahren nach spätestens $2\log(k)$ Merge-Schritten abgeschlossen und es bleibt lediglich ein einziges sortiertes Array $A$ übrig.

*Korrekturhinweis: Man kann auch zunächst annehmen dass $k$ eine Zweierpotenz ist und dann argumentieren, dass das Verfahren nur einen Mergeschritt mehr benötigt, wenn man $k$ auf die nächste Zweierpotenz aufrundet und entsprechend viele Dummyelemente mit Wert*

$\infty$ *einfügt. Es reicht sogar aus zu erwähnen dass höchstens ein konstanter Faktor mehr Schritte benötigt wird wenn $k$ keine Zweierpotenz ist. Wer überhaupt nicht darauf eingeht, dass $k_i$ ungerade sein könnte bekommt einen Punkt abgezogen.*

Das Mergen zweier Arrays $B, C$ dauert $\mathcal{O}(|B| + |C|)$. Damit benötigt jeder Merge Schritt insgesamt höchstens $\mathcal{O}\big(|A_1^i| + \ldots + |A_{k_i}^i|\big) = \mathcal{O}(km)$ Zeitschritte. Multipliziert mit der Anzahl der Merge-Schritte ergibt das $\mathcal{O}(km \log k)$

*Korrekturhinweis: Eine Analyse muss laut Aufgabenstellung nicht gemacht werden. Wenn der Algorithmus länger dauert gibt es aber entsprechend Abzug. Maximal einen Punkt gibt es für Algorithmen mit Laufzit $\mathcal{O}(km \log(km))$, weil das trivialierweise dem Hintereinanderschreiben der Listen $A_1, \ldots, A_k$ in eine Liste $A$ und Anwendung von Mergesort auf $A$ entspricht. Einen Punkt kann man für die Beschreibung eines "$k$-fach Merges" geben, bei der inital Pointer auf die ersten Arrayelemente gesetzt werden, das Maximum aus diesen bestimmt wird, und der entsprechende Pointer eine Stelle weiter gerückt wird. Das hat zwar Laufzeit $O(k^2 m)$ ist aber immerhin für $k \in o(\log m)$ besser als die triviale Lösung.*

(d) Wir erhalten $A'$ indem wir das größte Element am Ende von $A$ und das kleinste Element am Anfang von $A$ miteinander vertauschen. *(2 Punkte)*

Nach jedem Durchlauf der inneren Schleife von Bubblesort rückt das kleinste Element genau eine Stelle nach vorne. Damit benötigt die äußere Schleife mindestens $n-1 \in \Omega(n)$ Durchläufe bis das Array $A'$ wieder sortiert ist. *(3 Punkte)*

(e)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 13 | 47 | 59 | 39 | 28 | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

(f) Die Wahrscheinlichkeit, dass zwei Elemente $x$ und $y$ die gleiche Abfolge an Positionen haben, ist beim Doppelhashing geringer, da hierfür $h_1(x) = h_1(y)$ und $h_2(x) = h_2(y)$ gelten muss.

## Aufgabe 2: Textsuche *(9 Punkte)*

Gegeben sei der Suchtext $T = 15653212631$ der Länge $n = 11$ und das Muster $P = 21$ der Länge $m = 2$. Wir führen den Rabin-Karb Algorithmus mit Basis $b = 7$ und der Hashfunktion $h(x) := x \mod 8$ durch.

*Hinweis: Die Wahl der Basis $b = 7$ bedeutet, dass wir das Muster $P$ (sowie Auschnitte des Textes $T$) als Zahlen zur Basis 7 interpretieren. Beispielsweise entspricht $32$ der Zahl $3{\cdot}7^1 + 2{\cdot}7^0 = 23$.*

(a) Geben Sie für jeden Schritt $s \in \{0, \ldots, 9\}$ den Hashwert $h\big(T[s, s{+}1]\big)$ an. *(5 Punkte)*

*Hinweis: $T[s, s{+}1]$ ist das Teilwort bestehend aus dem $s$-ten und $(s{+}1)$-ten Zeichen von $T$.*

(b) Für welche $s \in \{0, \ldots, 9\}$ ist $h\big(T[s, s{+}1]\big) = h(P)$? *(4 Punkte)*

## Musterlösung

(a) Wir erhalten die folgenden Ergebnisse:

| $s$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $T[s, s{+}1]_{10}$ | 12 | 41 | 47 | 38 | 23 | 15 | 9 | 20 | 45 | 22 |
| $h\big(T[s, s{+}1]\big)$ | 4 | 1 | 7 | 6 | 7 | 7 | 1 | 4 | 5 | 6 |

*Korrekturhinweis: 0,5 Punkte Abzug für jeden Fehler in der letzten Zeile.*

(b) $s \in \{2, 4, 5\}$.

*Korrekturhinweis: 1,5 Punkte Abzug für jedes vergessene oder falsch eingefügte Element.*

# Aufgabe 3: Algorithmenanalyse *(15 Punkte)*

Der folgende Algorithmus erhält als Eingabe $k$ Arrays $A_1, \ldots, A_k$ der Größe $m$ und ein Array $B$ der Größe $k$. Die Arrays $A_i[0..m-1]$ enthalten ganze Zahlen in **sortierter** Reihenfolge. Das Array $B[0..k-1]$ enthält ganze Zahlen in **beliebiger Reihenfolge**.

**Algorithm 1** `algorithm(`$A_1, \ldots, A_k, B$`)`

```
x ← 0
for i ← 1 to k do
    j ← 0
    while j < m and A_i[j] < B[i−1] do
        j ← j + 1
    if A_i[j] ≠ B[i−1] then x ← x + 1

if x mod 2 = 0 then return True
else return False
```

(a) Geben Sie die Ausgabe von `algorithm(`$A_1, A_2, A_3, B$`)` für die folgende Eingabe an:

$$A_1 = [1, 4, 5, 6], A_2 = [1, 3, 4, 7], A_3 = [2, 4, 6, 9], B = [4, 2, 7].$$

*(3 Punkte)*

(b) Fassen Sie zusammen welche Aussage `algorithm` im Bezug auf die Eingabe $A_1, \ldots, A_k, B$ erzeugt. *(4 Punkte)*

(c) Geben Sie die **asymptotische** Laufzeit von `algorithm` im **worst case** in Abhängigkeit von $k$ und $m$ an. *(3 Punkte)*

(d) Beschreiben Sie einen Algorithmus, welcher die **gleiche Ausgabe** erzeugt wie `algorithm` und dessen Laufzeit $\mathcal{O}(k \log m)$ beträgt (oder beschreiben Sie eine entsprechende Modifikation von `algorithm`). *(5 Punkte)*

# Musterlösung

(a) `True` *(3 Punkte)*

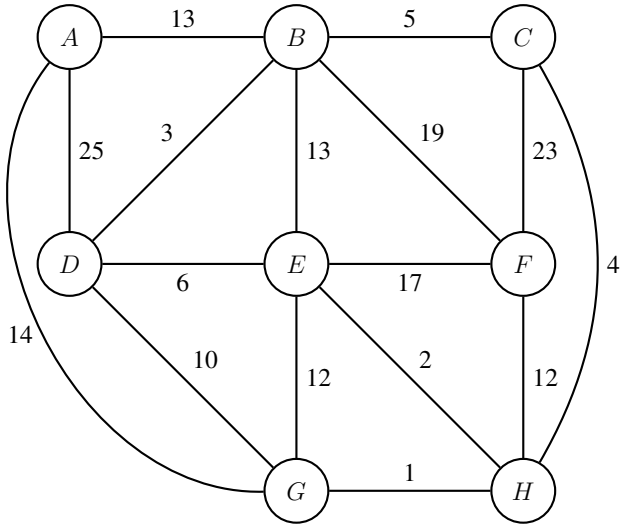
(b) `algorithm` zählt zunächst in wie vielen der Arrays $A_i$ ein Element vorkommt das jeweils *nicht* mit $B[i-1]$ übereinstimmt. Dann liefert `algorithm` den `True` zurück falls diese Anzahl gerade ist, sonst `False`. *(4 Punkte)*

(c) $\Theta(km)$ *Korrekturhinweis:* $\mathcal{O}(km)$ *ist ebenfalls in Ordnung.* *(3 Punkte)*

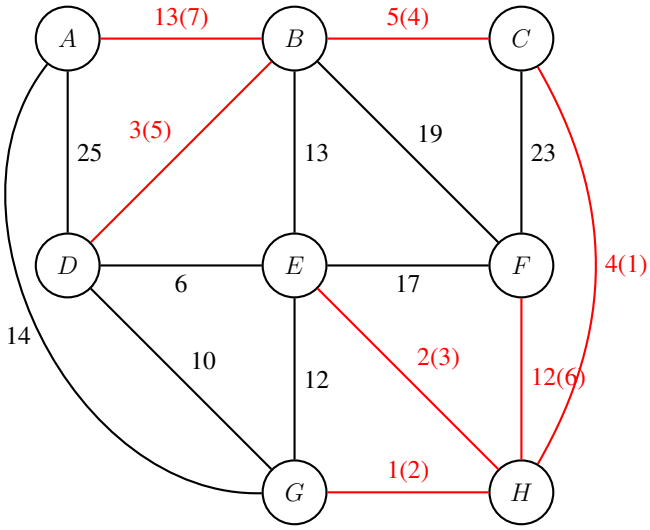
(d) Wir nutzen aus dass die Arrays $A_1, \ldots, A_k$ sortiert sind und modifizieren `algorithm` dahingehend dass wir statt der linearen Suche in der inneren Schleife eine binäre Suche durchführen (*Korrekturhinweis: Das ist schon ausreichend für die volle Punktzahl*). Dass heißt, wir springen jeweils zur Mitte des übrig gebliebenen Suchbereichs (inital das gesamte Array $A_i$) vergleichen das dortige Element mit $B[i-1]$ und machen mit der linken bzw. rechten Hälfte weiter wenn das gefundene Element größer bzw. kleiner war. *(5 Punkte)*

## Aufgabe 4: Minimaler Spannbaum *(7 Punkte)*

Führen Sie auf dem abgebildeten Graphen **Prim**'s Algorithmus zur Berechnung eines minimalen Spannbaums aus. Starten Sie mit Knoten C. Markieren Sie (falls farbig: nicht mit rot!) die Kanten, die am Ende im Baum sind und schreiben Sie neben die Kanten, in welcher Reihenfolge diese eingefügt werden.




## Musterlösung



# Aufgabe 5: Tankproblem *(18 Punkte)*

Ein Reisender möchte eine Strecke vom Startpunkt $a = 0$ zur Zielmarke $b \in \mathbb{N}$ mit seinem Auto zurücklegen. Mit vollem Tank kann das Auto eine **Strecke von $x \in \mathbb{N}$** zurücklegen. Zu Beginn der Reise ist der Tank voll. Auf der Strecke von $a$ nach $b$ liegen **$n$ Tankstellen** (mit Nummern $1, \ldots, n$) mit **aufsteigenden** Distanzen $t_1, \ldots, t_n \in \mathbb{N}$ zu $a$ die der Reisende kennt.

Der Reisende muss an einer Teilmenge $T \subseteq \{1, \ldots, n\}$ von Tankstellen anhalten um zu tanken. Dabei darf die Distanz zwischen zwei nacheinander besuchten Punkten aus $T \cup \{a, b\}$ jeweils höchstens $x$ sein, damit die Tankfüllung bis zum jeweils nächsten Ziel ausreicht.

(a) Geben Sie eine **Greedy-Strategie** um eine Auswahl $T$ von Tankstellen zu treffen, welche $|T|$ minimiert. *(2 Punkte)*

(b) Beweisen Sie, dass Ihre Strategie $|T|$ minimiert. *(7 Punkte)*

Es herrscht eine Ölkrise und an den Tankstellen haben sich Warteschlangen gebildet. Der Reisende hat sich informiert und kennt die **Wartezeit $w_i$** an jeder Tankstelle $i \in \{1, \ldots n\}$.

(c) Geben Sie einen effizienten Algorithmus nach dem **Prinzip des Dynamischen Programmierens** an, der eine Menge von Tankstellen $T \subseteq \{1, \ldots, n\}$ ermittelt, welche die aufsummierte Wartezeit $W = \sum_{j \in T} w_j$ minimiert. Bestimmen Sie die Laufzeit Ihres Algorithmus. *(9 Punkte)*

# Musterlösung

(a) **Als Text:** Sei $t_0 := 0$, $j := 0$ und $T = \emptyset$. (Schleife:) Solange $t_j + x < b$ (wir erreichen wir die Zielmarke $b$ mit der aktuellen Tankfüllung noch nicht) fügen wir die weiteste Tankstelle $t_k$ ($k > j$) die vom aktuellen Standort $t_j$ noch innerhalb unserer Reichweite $x$ ist zu $T$ hinzu setzen $j := k$ und führen die nächste Iteration der Schleife durch. Am Ende geben wir $T$ zurück.

*Korrekturhinweis: Es darf davon ausgegangen werden, dass die Abstände zwischen Tankstellen kleiner als $x$ sind. Deshalb kann, aber muss nicht erwähnt werden: Falls weder eine Tankstelle noch das Ziel von der aktuellen Position aus erreichbar sind, kann man* `False` *o.ä. zurückgeben.*

**Algorithm 2** `Greedy-Refueling`$(x, t_1, \ldots, t_n, b)$

$j \leftarrow 0$; $t_0 \leftarrow 0$; $T \leftarrow \emptyset$
**while** $t_j + x < b$ **do**
  **if** there exists $k > j$ with $t_j + x \geq t_k$ **then**
    $j \leftarrow \underset{k>j,\, t_j+x \geq t_k}{\arg\max}\ t_k$
    $T \leftarrow T \cup \{j\}$
  **else return** `False` ▷ *Korrekturhinweis: Fallabfrage nicht notwendig*
**return** $T$

(b) Sei $T^* = \{i_1, \ldots, i_\ell\}$ eine Tankstrategie, sodass $|T^*|$ minimal ist und sei $T = \{j_1, \ldots, j_m\}$ die Lösung unserer Greedy Strategie (jeweils geordnet nach Reihenfolge ihres Auftretens auf der Route). Sei die $k$-te Tankstop der erste, sodass $T^* \ni i_k \neq j_k \in T$, d.h. der erste Tankstop entlang der Route, der sich bei den Tankstrategien $T^*$ und $T$ unterscheidet.

Nach unserer Greedy Wahl von $j_k \in T$ ist $t_{j_k} > t_{i_k}$ und $t_{j_{k-1}} + x \geq t_{j_k}$ und $t_{j_{k-1}} + x < b$. Da die vorherige Tankstelle $j_{k-1} = i_{k-1}$ in $T$ bzw. $T^*$ übereinstimmen und da $t_{i_k} > t_{j_k}$, können wir in $T^*$ die Tankstelle $j_k$ mit der Tankstelle $i_k$ ersetzen. Die neue Lösung $T^{(1)} := (T^* \setminus \{i_k\}) \cup \{j_k\}$ ist ebenfalls zulässig und minimal da $|T^*| = |T^{(1)}|$.

Falls $T = T^{(1)}$ ist $|T| = |T^*|$ und somit ebenfalls minimal. Sonst wiederholen wir den obigen Austausch mit $T^* = T^{(1)}$ um Lösungen $T^{(2)}, T^{(3)}, \ldots$ zu erhalten, solange bis das Ergebnis mit $T$ übereinstimmt. Da wir stets gleich große Lösungen erhalten ist schließlich $|T^*| = |T^{(1)}| = |T^{(2)}| = \ldots = |T|$.

(c) Wir definieren $W_i$ als minimale Wartezeit die ausgehend vom Standort $t_i$ notwendig ist um (unter der Annahme eines *bereits vollen Tanks*) das Ziel $b$ zu erreichen. Für $i = 0$ sei $t_0 := 0$ der Startpunkt. Sei $T_i \subseteq \{1, \ldots, n\}$ die Menge der Tankstellen die der Reisende ab $t_i$ ansteuern muss um die minimale Wartezeit $W_i$ zu verwirklichen.

Wir definieren $W_i$ rekursiv wie folgt: Im Basisfall ist $t_i + x \geq b$ und somit $W_i := 0$ und $T_i = \emptyset$ (wir können direkt zum Ziel fahren ohne weiter tanken zu müssen). Sonst setzen wir $j := \arg\min_{i<j\leq n,\, t_j+x \geq t_i} \big(W_j + w_j\big)$ und $W_i := W_j + w_j$ und $T_i := T_j \cup \{j\}$. Das gewünschte Ergebnis ist $W_0 = W$ und $T_0 = T$. Diese Rekursion können wir für das übliche Algorithmendesign nach dem Prinzip des dynamischen Programmierens benutzen. Nach Ausführung von `Dynamic-Refueling(0)` liegt das Ergebnis in `T`$[0]$ vor. *(7 Punkte)*

**Algorithm 3** `Dynamic-Refueling`$(i)$ ▷ globale dictionaries `W` und `T`

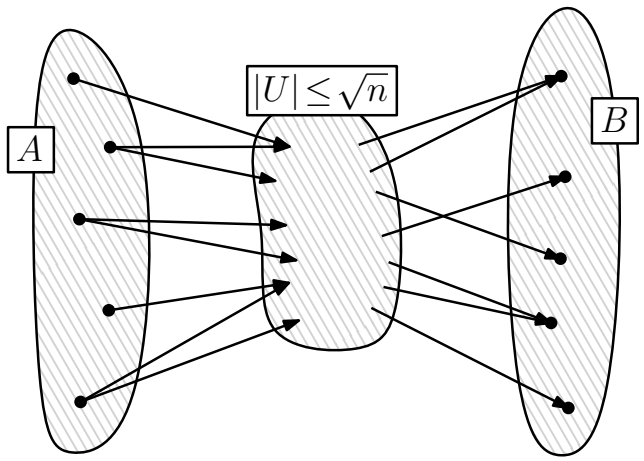
**if** $t_i + x \geq b$ **then**
  `W`$[i] \leftarrow 0$; `T`$[i] \leftarrow \emptyset$
  **return** $0$
**else if** `W`$[i]$ not empty **then** ▷ *Zwischenergebnis bereits berechnet*
  **return** `W`$[i]$
**else** $j \leftarrow \underset{i<j\leq n,\, t_j+x \geq t_i}{\arg\min}\ \big($`Dynamic-Refueling(`$j$`)` $+ w_j\big)$
  `W`$[i] \leftarrow$ `W`$[j] + w_j$; `T`$[i] \leftarrow$ `T`$[j] \cup \{j\}$
  **return** `W`$[i]$

*Korrekturhinweis: Zwei Punkte Abzug für die Berechnung von $W$ aber nicht $T$.*

Die Laufzeit eines Rekursionsschrittes `Dynamic-Refueling(`$i$`)` ist $\mathcal{O}(n)$ für die Bestimmung des $\arg\min$ unter Missachtung der rekursiven Unteraufrufe. Wir berechnen jeden Rekursionsschritt `Dynamic-Refueling(`$i$`)`, $i = 1, \ldots, n$ höchstens ein mal bevor das Ergebnis jeweils im Dictionary vorliegt. Die Gesamtlaufzeit ist also $\mathcal{O}(n^2)$. *(2 Punkte)*

# Aufgabe 6: Kürzeste Pfade *(21 Punkte)*

Gegeben sei eine Klasse $\mathcal{G}$ von **gerichteten, gewichteten** Graphen. Graphen $G = (V, E, w)$ der Klasse $\mathcal{G}$ haben **positive Kantengewichte** und deren Knoten $V$ ($|V| = n$) bestehen aus **drei disjunkten Teilmengen $A$, $B$ und $U$**, so dass folgende Eigenschaften erfüllt sind. **Knoten in $A$ haben nur ausgehende Kanten** zu Knoten in $U$ und **Knoten in $B$ haben nur eingehende Kanten** von Knoten in $U$. Die Knoten in $U$ können durch beliebige Kanten verbunden sein, allerdings **enthält die Menge $U$ höchstens $\sqrt{n}$ Knoten**. Eine Verbildlichung:




Optimieren Sie die folgenden, aus der Vorlesung bekannten Algorithmen zur Berechnung der Distanz zwischen **allen Paaren von Knoten**, **für Graphen der Klasse $\mathcal{G}$**. Argumentieren Sie jeweils, warum der optimierte Algorithmus **korrekt** ist und drücken Sie die asymptotische **Laufzeit** als Funktion von $n$ aus.

(a) Bellman-Ford Algorithmus. *(7 Punkte)*

(b) Das Verfahren mittels Matrix-Multiplikationen in der Min-Plus-Algebra. *(7 Punkte)*

(c) Floyd-Warshall-Algorithmus. *(7 Punkte)*

*Hinweise: Sie können davon ausgehen, dass die Aufteilung in die Mengen $A$, $B$ und $U$ bekannt ist. Es genügt die Modifikation der genannten Algorithmen ausreichend genau zu beschreiben, Sie müssen keinen Pseudocode angeben.*

# Musterlösung

Für die Optimierung der drei Wegsuchalgorithmen benutzen wir die folgende *Beobachtung: Da alle kürzesten Pfade in Graphen der Klasse $\mathcal{G}$ nur innere Knoten aus der Menge $U$ haben können, haben kürzeste Pfade höchstens $|U|+1 = \sqrt{n}+1$ Kanten.*

(a) Wir ändern lediglich die äußere Schleife des Bellman-Ford Algorithmus (mit einem Startknoten $s$), sodass diese nur $\lfloor\sqrt{n}\rfloor+1$ Iterationen durchläuft (statt $n-1$). Da der Bellman-Ford Algorithmus nach $i$ Durchläufen der äußeren Schleife alle kürzesten Pfade findet die höchstens $i+1$ Kanten haben, reichen aufgrund der obigen Beobachtung $\lfloor\sqrt{n}\rfloor+1$ Durchläufe aus um die Distanzen zu $s$ korrekt zu berechnen. Die Laufzeit von Bellman-Ford verringert sich auf $\mathcal{O}(\sqrt{n}|E|) = \mathcal{O}(n^{5/2})$. Da wir Bellman-Ford insgesamt $n$ mal ausführen müssen um die Distanzen zwischen allen Knotenpaaren zu ermitteln, erhöht sich die Laufzeit auf $\mathcal{O}(n^{3/2}|E|) = \mathcal{O}(n^{7/2})$. Der Speedup ist also $\sqrt{n}$.

(b) Wir benutzen wie zuvor die obige Beobachtung um das Verfahren zu optimieren. Der Iterationsparameter $t$ der äußeren Schleife muss nur noch bis $\lceil\log(\sqrt{n})\rceil$ laufen (statt $\lceil\log n\rceil$) damit wir die korrekte Ausgabe erhalten. Der modifizierte Algorithmus funktioniert für Graphen der Klasse $\mathcal{G}$, da nach allen Durchläufen der Schleife die Distanzen von kürzesten Pfaden mit höchstens $2^{\lceil\log(\sqrt{n})\rceil}+1 \geq \sqrt{n}+1$ Kanten korrekt berechnet sind, was nach obiger Beobachtung für alle kürzesten Pfade der Fall ist. Die asymptotische Laufzeit ist $\mathcal{O}(n^3\log(\sqrt{n})) = \mathcal{O}(n^3\log n)$, ändert sich also nicht. Wir erhalten dennoch einen konstanten Speedup von $\frac{n^3\cdot\log n}{n^3\cdot\log(\sqrt{n})} = 2$.

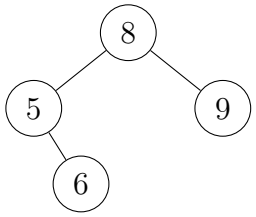
(c) Ähnlich wie bei Bellman-Ford können wir die Anzahl der Durchläufe bei Floyd-Warshall auf $\lfloor\sqrt{n}\rfloor$ absenken. Wir benutzen dazu, dass die Menge $U$ bekannt ist (Hinweis). Seien ohne Einschränkung $U = \{1,\ldots,\ell\}$ mit $\ell \leq \sqrt{n}$ die Nummern der Knoten von $U$. Nach dem $k$-ten Durchlauf der Hauptschleife des Algorithmus, sind die Distanzen zwischen allen kürzesten Pfaden die lediglich die Knoten mit Nummern $1,\ldots,k$ als innere Knoten verwenden, korrekt berechnet. Da aufgrund obiger Beobachtung lediglich die Knoten $1,\ldots,\ell$ mit $\ell \leq \sqrt{n}$ innere Knoten von kürzesten Pfaden sein können, reicht es die Hauptschleife bis $\lfloor\sqrt{n}\rfloor$ laufen zu lassen um die korrekten Distanzen zwischen allen Knotenpaaren zu erhalten. Die asymptotische Laufzeit ist dann $\mathcal{O}(n^{5/2})$ und der Speedup ist $\sqrt{n}$.

*Korrekturhinweis: Alle korrekten Modifikationen die einen Speedup geben, bzw. für gewisse Eingaben einen Speedup geben könnten, werden anteilig bewertet, selbst wenn dies unter dem Speedup der Musterlösung bleibt. Es können folgende Unterscheidung gemacht werden (Liste nicht erschöpfend):*

- *Ist der Speedup "additiv variabel" (gemeint ist $T_{original} - T_{mod} = x$ für $x \in \Omega(n)$ für **manche** Eingaben $G$) gibt das je nach Speedup höchstens zwei Punkte mit korrektem Algorithmus und korrekter Analyse (ohne letzteres 1).*
- *Ist der Speedup "multiplikativ variabel" (gemeint ist $T_{original}/T_{mod} = x$ für $x \in \omega(1)$ für **manche** Eingaben $G$) gibt das je nach Speedup $x$ bis zu drei Punkte mit korrektem Algorithmus und korrekter Analyse (ohne letzteres 1,5).*
- *Ist der Speedup "multiplikativ" (gemeint ist $T_{original}/T_{mod} = x$ für $x \in \Omega(1)$ für **alle** Eingaben $G$) gibt das je nach Speedup $x$ die volle Punktzahl mit korrektem Algorithmus und korrekter Analyse (ohne letzteres die Hälfte).*

## Aufgabe 7: Binäre Suchbäume *(27 Punkte)*

(a) Gegeben sei der folgende binäre Suchbaum:




Geben Sie **alle Folgen** von `insert`$(key)$ Operationen an, welche diesen Baum erzeugen. Entspricht einer dieser Folgen der **Pre-Order**-Traversierung? Entspricht einer dieser Folgen der **Post-Order**-Traversierung? Wenn ja, welche? *(5 Punkte)*

(b) Zeigen Sie folgende Eigenschaft binärer Suchbäume: Wenn ein Knoten zwei Kinder hat, dann hat sein Vorgänger kein rechtes Kind und sein Nachfolger kein linkes Kind. *(6 Punkte)*

(c) Seit $T$ ein binärer Suchbaum, bei dem die Knoten statt pointer zum **parent**-Knoten jeweils einen pointer zu ihrem **Vorgänger** und **Nachfolger** haben, d.h. für alle Knoten $u$ gibt es die Attribute $u.pred$ und $u.succ$.

Beschreiben Sie einen (möglichst effizienten) Algorithmus, der von einem Knoten $u$ den parent-Knoten zurückgibt. Begründen Sie die Korrektheit Ihres Algorithmus und analysieren Sie die Laufzeit (in Abhängigkeit von der Tiefe des Baums). *(8 Punkte)*

*Hinweis: Überlegen Sie sich zunächst wie man vorgehen würde, wenn man wüsste, ob $u$ das linke oder das rechte Kind seines parents ist (oder ob $u$ gar keinen parent hat).*

(d) Ein binärer Suchbaum heiße **perfekt balanciert**, wenn für jeden Knoten $u$ sich die Anzahl der Knoten im linken Teilbaum von $u$ von der Anzahl der Knoten im rechten Teilbaum von $u$ um höchstens $1$ unterscheidet. Beschreiben Sie einen (möglichst effizienten) Algorithmus, der testet, ob ein binärer Suchbaum perfekt balanciert ist. Analysieren Sie die Laufzeit Ihres Algorithmus. *(8 Punkte)*

# Musterlösung

(a) (i) 8 5 6 9

(ii) 8 5 9 6

(iii) 8 9 5 6

(i) entspricht der pre-order Traversierung. Keine entspricht der post-order Traversierung.

(b) Wenn ein Knoten $x$ ein linkes Kind hat, dann ist sein Vorgänger der Knoten mit dem größten Schlüssel im linken Teilbaum von $x$. Dieser hat kein rechtes Kind, da dies auch im linken Teilbaum von $x$ wäre und einen noch größeren Schlüssel hätte.

Wenn ein Knoten $x$ ein rechtes Kind hat, dann ist sein Nachfolger der Knoten mit dem kleinsten Schlüssel im rechten Teilbaum von $x$. Dieser hat kein linkes Kind, da dies auch im rechten Teilbaum von $x$ wäre und einen noch kleineren Schlüssel hätte.

(c) Sei $\min(x)$ der Knoten mit kleinstem und $\max(x)$ der Knoten mit größtem Schlüssel des Teilbaums mit Wurzel $x$. Diese kann man (ohne einen parent-pointer) in $\mathcal{O}(\text{Tiefe})$ finden. Falls $x = x.\textit{parent.left}$, so ist $x.\textit{parent} = \max(x).\textit{succ}$. Falls $x = x.\textit{parent.right}$, so ist $x.\textit{parent} = \min(x).\textit{pred}$. Der Algorithmus lautet also

**Algorithm 4** `findparent(`$x$`)`

```
if x = max(x).succ.left then
    x.parent = max(x).succ
else if x = min(x).pred.right then
    x.parent = min(x).pred
else
    x.parent =none
```

(d) Der folgende Algorithmus bekommt als Eingabe einen Schlüssel $x$ und gibt ein Tupel $(y, z)$ aus, wobei $y$ entweder true oder false ist, je nachdem ob der Teilbaum mit Wurzel $x$ perfekt balanciert ist, und $z$ die Anzahl der Elemente im Teilbaum mit Wurzel $x$ ist (wird für die Rekursion benötigt).

**Algorithm 5** A($x$)

```
if x == None then
    return (true,0)
else
    if A(x.left)[0] == true & A(x.right)[0] == true & |A(x.left)[1]-A(x.right)[1]|≤ 1 then
        return (true,A(x.left)[1]+A(x.right)[1]+1)
    else
        return (false,0)
```

Der Algorithmus wird für jeden Knoten einmal aufgerufen und ein Durchgang (ohne Rekursion) dauert $\mathcal{O}(1)$. Die Laufzeit beträgt also $\mathcal{O}(n)$.