# Algorithmen und Komplexität
# Musterlösung Klausur Winter 2018

## Lösungsvorschlag zu Aufgabe 1

(a) Falsch, da man in der Liste der Nachbarn von $u$ prüfen muss, ob $v$ vorhanden ist (oder umgekehrt) und dies im Allgemeinen nicht in konstanter Zeit möglich ist.

(b) Falsch. Betrachte zum Beispiel $f(n) = 1$ und $g(n) = n$.

(c) Wahr, da $(\log n)^{\log n} = e^{\log\log n \cdot \log n} = \Omega(e^{5 \log n}) = \Omega(n^5)$.

(d) Wahr. Zusammenhang kann mit Breitensuche in Zeit $O(|V| + |E|)$ getestet werden. Aus der Voraussetzung folgt $|E| \leq 5|V|$, also kann Zusammenhang in Zeit $O(|V|)$ getestet werden.

(e) Wahr. $T$ kann 9 Schlüssel haben (wenn alle internen Knoten 3 Kinder haben) und $T'$ kann 8 Schlüssel haben (wenn alle internen Knoten 2 Kinder haben).

(f) Falsch. Man würde effiziente Implementationen der Operationen EXTRACT-MIN und DECREASE-KEY benötigen. Die Union-Find Datastruktur ünterstützt diese Operationen nicht.

(g) Wahr. Am Ende von EXTRACT-MIN wird CONSOLIDATE ausgeführt wird. Deshalb haben alle Knoten in der Wurzelliste unterschiedlichen Rank. Die Aussage folgt aus Korollar 4.14.

(h) Wahr, da 3-$\mathcal{SAT}$ $\mathcal{NP}$- vollständig.

(i) Falsch. Geschicktes oder ungeschicktes ausprobieren zeigt dies.

(j) Wahr. Dies ist genau die Definition einer berechenbaren Sprache.

## Lösungsvorschlag zu Aufgabe 2

(a) Falsch. Betrachte einen Pfad bestehend aus zwei Kanten (und drei Knoten), die beide das Gewicht 1 haben. Es gibt nur einen Spannbaum (also ist der minimal Spannbaum eindeutig). Definiert man $W$ als Menge der Endpunkte des Pfades so haben beide Kanten zwischen $W$ und $V \setminus W$ dasselbe Gewicht.

(b) Falsch. Betrachte zum Beispiel $n = 3$. Wenn Tiefen- und Breitensuche beim selben Knoten $v$ beginnen, so fügt Breitensuche die beiden Kanten bei $v$ hinzu, wohingegen Tiefensuche nur eine Kante bei $v$ hinzufügt. Die beiden Spannbäume sind also verschieden.

(c) Wahr. Da jeder Knoten in einem Dreieck enthalten ist, hat jeder Knoten mindestens Grad 2. Aus der Formel (siehe Vorlesung) $|E| = \left(\sum_{v \in V} \deg(v)\right)/2$ folgt daher

$$|E| = \Big(\sum_{v \in V} \deg(v)\Big)/2 \geq 2|V|/2 = |V|.$$

(d) In der Vorlesung haben wir gesehen, dass beide Probleme $\mathcal{NP}$-vollständig sind. Insbesondere ist KNAPSACK$\in \mathcal{NP}$ und ERFÜLLBARERSCHALTKREIS $\mathcal{NP}$-schwer, woraus die Aussage folgt.

## Lösungsvorschlag zu Aufgabe 3

(a) Ein Graph ist zweifärbbar genau dann wenn er bipartit ist. Überprüfen ob ein graph bipartit ist können wir mittels BFS/DFS in Zeit $O(|V| + |E|) = O(|V^2|)$ (aus der Serie 3 Aufgabe 1). Nun schlagen wir folgenden Algorithmus vor. Für jeden Knoten in $V$, überprüfe ob $G \backslash \{v\}$ bipartit ist. Falls ja für irgend einen Knoten, dann ist der Graph auch dreifärbbar mit maximal einem blauen knoten, andernfalls nicht. Als Pseudocode

```
Given: Graph G = (V, E) ;
for all v ∈ V do
  if G\{v} is bipartite then
    return true.
  end if
end for
return false.
```

Laufzeit: Wir durchlaufen die for-Schlaufe maximal einmal für jeden Knoten, also $O(|V|)$. In der vorschlaufe berechnen wir den neuen Graphen $G \backslash \{v\}$, was maximal $O(|V| + |E|)$ Zeit braucht, und testen ob bipartit braucht auch höchstens $O(|V| + |E|)$. Also benötigt der Algotithmus maximal Zeit $O(|V|(|V| + |E|)) = O(|V^3|)$.

Korrektheit: Falls der Algorithmus true ausgibt, gibt es einen Knoten ohne den man den restlichen graphen 2-färben kann. Also wenn man diesen Knoten blau färbt kann man die restlichen mit rot und grün färben und erhält so eine gültige Färbung wie in der Aufgabenstellung. Umgekehrt falls es eine gültige Färbung gibt, dann ist höchstens ein Knoten blau gefärbt und entsprechend muss der graph ohne diesen Knoten 2-färbbar also bipartit sein. Man bemerke noch falls es eine 2-Färbung des Graphen gibt (also mit keinem blauen knoten) dann gibt es trotzdem auch eine 3-Färbung indem man einen beliebigen Knoten einfach blau färbt.

(b) Mit dem Algorithmus $\frac{n+1}{2}$-Select kann man den Median $m$ berechnen. Nun erstellt man ein zweites Array $B$ der Länge $n$ und setzt für alle $i \in [n]$, $b_i = |m - a_i|$. Nun kann man wiederum $k$-Select auf das Array $B$ anwenden um $t$ zu erhalten. Zum Schluss gibt man $a_i$ aus genau dann wenn $b_i \leq t$. Als Pseudocode

```
Given: Array A = [a_1, .., a_n] and k ;
m ← Select(A, (n+1)/2)
B ← Array[n]
for i = {1, ..., n} do
  b_i ← |m − a_i|
end for
t ← Select(B, k)
C ← ∅
for i = {1, ..., n} do
  if b_i ≤ t then
    C = C ∪ {a_i}
  end if
end for
return C
```

Laufzeit: Select haben wir in der Vorlesung bewiesen hat eine Laufzeit von $O(n)$. Alle for-Schlaufen laufen $n$ mal und alles in der Schlaufe können wir in konstanter Zeit machen. Also ist die gesammte Laufzeit $O(n)$.

Korrektheit: Da $n$ ungerade berechnet $Select(A, \frac{n+1}{2})$ korrekterweise den Median. Im Array $B$ haben wir jeweils die differenz zum Median. Also wählt $Select(B, k)$ die $k$-kleinste Differenz zum Median. Entsprechend geben wir alle Zahlen aus deren Differenz kleiner oder gleich $Select(B, k)$ ist indem wir mit jedem Element von $B$ vergleichen.

## Lösungsvorschlag zu Aufgabe 4

Wir lösen die Aufgabe nach dem Standardschema für das Nachweisen von $\mathcal{NP}$-Vollständigkeit. Als erstes zeigen wir, dass die Sprache FOOTBALLCARDCOLLECTOR in $\mathcal{NP}$ liegt. Dazu betrachten wir eine Instanz $x$, die in der Sprache liegt. Das Zertifikat $w$ seien die $k$ Päckchen $P_1, \ldots, P_k$, die alle $n$ Spieler enthalten. Da es sich dabei um eine Teilmenge aller Päckchen handelt, ist $|w|$ polynomiell beschreingt in $|x|$. Um nachzuweisen, dass diese Instanz tatsächlich in unserer Sprache liegt, können wir z.B. für jeden Fussballer die $k$ Päckchen durchgehen und schauen, ob er irgendwo enthalten ist. Dies verursacht Laufzeit $O(n \cdot |w|) = O(|x|^2)$, also liegt unsere Sprache in $\mathcal{NP}$.

Jetzt zeigen wir VERTEXCOVER $\leq_P$ FOOTBALLCARDCOLLECTOR, denn da VERTEXCOVER gemäss Ferien Serie $\mathcal{NP}$-vollständig ist, folgt daraus auch die $\mathcal{NP}$-Vollständigkeit vom zweiten Problem. Wir müssen also eine polynomiell berechenbare Funktion $f$ finden, so dass $x \in$VERTEXCOVER genau dann wenn $f(x) \in$FOOTBALLCARDCOLLECTOR.

Sei $(G, k)$ eine Eingabe für VERTEXCOVER, wobei $G = (V, E)$ ein Graph ist. Für die Reduktion führen wir jetzt für jede Kante $e \in E$ einen Spieler $Y_e$ und für jeden Knoten $v \in V$ ein Päckchen $P_v$ ein, so dass $P_v$ genau diejenigen Spieler $Y_e$ enthält, für welche im Graphen $G$ die Kante $e$ inzident zum Knoten $v$ ist. Das $k$ lassen wir unverändert. Jeder Fussballer wird nur 2 mal in ein Päckchen gelegt, also schauen wir uns jede Kante 2 mal an. Somit können wir die Reduktion $f$ in Laufzeit $O(|V| + |E|)$ konstruieren, was offensichtlich polynomiell ist.

Es bleibt zu zeigen, dass unsere Reduktion die gewünschte Äquivalenzeigenschaft erfüllt. Sei $G = (V, E)$ ein gegebener Graph, auf den wir unsere Reduktion abbilden. Falls $G$ ein Vertex-Cover der Grösse $k$ aufweist, so gibt es eine Menge $S = \{v_1, \ldots, v_k\}$ von $k$ Knoten, so dass jede Kante im Graphen $G$ zu mindestens einem Knoten in $S$ inzident ist. Sei nun $Y_e$ ein beliebiger Spieler. Da $S$ ein Vertex-Cover ist, gibt es ein Knoten $v_i \in S$ so dass $e$ inzident zu $v_i$ ist. Nach Konstruktion ist dann Spieler $Y_e$ im Päckchen $P_{v_i}$ enthalten. Wir sehen, dass die $k$ Päckchen $P_{v_1}, \ldots, P_{v_k}$ alle Spieler enthalten.

Falls es umgekehrt in der konstruierten Instanz $f(G)$ für das Sammlerproblem $k$ Päckchen $P_1, \ldots, P_k$ gibt, die alle Spieler enthalten, dann entspricht per Konstruktion jedes dieser Päckchen $P_i$ einem Knoten $v_i$. Sei $e \in E$ eine beliebige Kante im ursprünglichen Graphen. Dann ist der Spieler $Y_e$ in mindestens einem Päckchen $P_i$ enthalten, gemäss Konstruktion ist folglich $e$ inzident zu $v_i$. Also bilden die Knoten $v_1, \ldots, v_k$ ein Vertex-Cover, und somit $G \in$VERTEXCOVER.

## Lösungsvorschlag zu Aufgabe 5

Sei $z_i$ der maximale Profit der an den Tagen $i, \ldots, n$ erlernt werden kann. Definiere $z_{n+2} = z_{n+1} = 0$, da an den Tagen $n+1$ und $n+2$ nicht gelernt werden kann. Betrachte nun den Tag $i \in \{1, \ldots, n\}$. Um $z_i$ zu maximieren ist es sicher besser wenig als gar nicht zu lernen (da $a_i > 0$ und in beiden Varianten an den folgenden Tagen gleichviel gelernt werden kann). Falls Annika am Tag $i$ wenig lernt so kann sie auch am Tag $i+1$ lernen und es gilt $z_i = a_i + z_{i+1}$. Falls Annika am Tag $i$ viel lernt so muss sie am Tag $i+1$ Pause machen und es gilt $z_i = b_i + z_{i+2}$. Da einer der beiden Fälle eintritt gilt für $i = n, \ldots 1$ folgende Rekursionsformel:

$$z_i = \max\{a_i + z_{i+1}, b_i + z_{i+2}\}$$

Wir berechnen $z_i$ für $n$ bis 1. $z_1$ ist dann der maximale Profit, den wir mit folgendem Algorithmus berechnen können:

$z_{n+1} \leftarrow 0$ ; $z_{n+2} \leftarrow 0$ ;
**for** $i = n$ to 1 **do**
  $z_i = \max\{a_i + z_{i+1}, b_i + z_{i+2}\}$
**end for**
**return** $z_1$ ist maximale Profit.

Um die Lernstrategie auszugeben, müssen wir Rückwärtslaufen und schauen was jeweils als letztes zum Wert $z_i$ addiert wurde.

Initialisiere array der Länge $n$: $x[1 \ldots n]$ ;
$i \leftarrow 1$;
**while** $i \leq n$ **do**
  **if** $z_i = a_i + z_{i+1}$ **then**
    $x_i \leftarrow$ wenig ;
    $i \leftarrow i + 1$;
  **else**
    $x_i \leftarrow$ viel ;
    $x_{i+1} \leftarrow$ Pause ;
    $i \leftarrow i + 2$;
  **end if**
**end while**
**return** $x[1 \ldots n]$.

*Korrektheit:* Die Rekursionsformel wurde oben begründet. Dass die optimale Lernstrategie ausgegeben wird ist dann offensichtlich.

*Laufzeit:* Laufzeit ist $O(n)$ da jede Iteration der beiden for-schleifen in konstanter Zeit ausgeführt werden kann.

*Bemerkungen:* Diese Aufgabe kann auf viele verschiedenen Arten gelöst werden. Wenn wir die Zeit nicht umdrehen wie oben, dann brauchen wir zusätzliche Informationen was am Tag $i$ gemacht wurde. Zum Beispiel sei $A_i$ bzw. $B_i$ der maximale Profit bis Tag $i$, wenn Annika am Tag $i$ wenig bzw. viel lernt. Dann gilt folgende Rekursion:

$$A_i = a_i + \max\{A_{i-1}, B_{i-2}\}$$
$$B_i = b_i + \max\{A_{i-1}, B_{i-2}\}$$

Eine andere Möglichkeit ist $f_i$ als maximalen Profit bis Tag, wenn am Tag $i$ wenig gelernt wird, zu definieren. Dann gilt folgende Rekursion:

$$f_i = a_i + \max\{f_{i-1}, f_{i-2} - a_{i-2} + b_{i-2}\}$$

## Lösungsvorschlag zu Aufgabe 6

Wir bemerken zuerst, dass es höchstens einen berühmten Knoten im Graph $G$ geben kann. Diese Erkenntnis führt nun zur folgenden Idee: Wir finden in linearer Zeit den einzigen möglichen Kanditaten $w \in V$ und testen dann, ob unser Kandidat auch tasächlich berühmt ist. Diese Idee wird in folegendem Algorithmus umgesetzt:

**Algorithmus:**

Sei $V = \{v_1, v_2, \ldots, v_n\}$.
Setze $k := 1$.
**for all** $i = 2$ to $n$ **do**
  **if** $(v_k, v_i) \in A$ **then**
    $k \leftarrow i$
  **end if**
**end for**
**for all** $i = [n] \setminus \{k\}$ **do**
  **if** $(v_k, v_i) \in E$ oder $(v_i, v_k) \notin A$ **then**
    **return** Kein berühmter Konten
  **end if**
**end for**
**return** $v_k$ ist der berühmte Knoten.

**Korrektheitsbeweis:**
Offensichtlich terminiert der Algorithmus und gibt die gewünschte Antwort aus. Wir brauchen nur noch zu zeigen, dass die Antwort Korrekt ist.

Zuerst überlegen wir uns, dass die Knoten $v_j$ mit $j \neq k$ sicherlich nicht berühmt sein können. Sei $K \subset V \setminus \{v_k\}$ die Menge der Knoten, welche einmal als Kandiat gehandelt wurden. Für alle Knoten in $K$ gilt nun, dass sie eine eingehende Kante besitzen (sie wurden ja als Kandidat abgelöst). Alle anderen Knoten besitzen aber eine ausgehende Kante, da sie nie als potenzieller Kandidat gehandelt wurden. Folglich kann keiner dieser Knoten berühmt sein.

Nachdem wir einen Kandidat $v_k$ gefunden haben, können wir einfach testen, ob alle eingehneden Kanten da sind und ob keine ausgehend Kante vorhanden ist (zweite For-Schlaufe). Gilt dies nicht, gibt es keinen berühmten Knoten. Sonst ist $v_k$ der berühmte Knoten und wird auch vom Algorithmus ausgegeben.

**Laufzeitanalyse:**
In den beiden For-Schlaufen greifen wir auf ein, repektive zwei, Elemente der Adjazenzmatrix $M$ zu und testen, ob diese Elemente gleich 1 sind oder nicht. Dies geschieht in konstanter Zeit. Somit brauchen wir für beide For-Schlaufen $O(n)$ Zeit, weshalb der ganze Algorithmus in Zeit $O(n)$ terminiert.