ETH Zürich
Institut für Theoretische Informatik
Prof. Dr. Angelika Steger, Dr. Johannes Lengler
Übungsleitung: Florian Meier

HS 2019

Algorithmen und Komplexität
Lösungsskizze Ferienserie

Lösungsvorschlag zu Aufgabe 1

Es bezeichne $[i] = \{1, \dots, i\}$. Es sei b_i die maximale Summe bis zum i-ten Element, welche das i-te Element enthält, das heisst, $b_i = \max_{k \in [i]} \sum_{j=k}^{i} a_j$.

Es gilt

$$\begin{aligned} b_i &= \max_{k \in [i]} \sum_{j=k}^{i} a_j \\ &= \max\{\max_{k \in [i-1]} \sum_{j=k}^{i} a_j, a_i\} \\ &= a_i + \max\{b_{i-1}, 0\} \end{aligned}$$

Mit Hilfe dieser Rekursionsformel erstellen wir folgendes dynamisches Program. Wir initialisieren $b_1 = a_1$ und berechnen b_i für $i = 2, \dots, n$ mit der Formel. Anschliessend suchen wir den maximalen Wert b_i, dies gibt uns den Wert der maximalen Summe und gleichzeitig den Index k. Um auch den Index ℓ zu finden, setzen wir $j = k$ und solang $b_j \neq a_j$ verringern wir j um 1 (im Falle $b_j = a_j$ wurde der linke Rand ℓ der Summe b_k erreicht).

Korrektheit folgt aus der Korrektheit der Rekursionsformel.

Laufzeit: Die For-Schleife zur Berechnung der b_i benötigt $\mathcal{O}(n)$, das suchen des maximalen b_i auch $\mathcal{O}(n)$ und das finden von Index ℓ ebenfalls $\mathcal{O}(n)$. Daraus ergibt sich die gewünschte Laufzeit $\mathcal{O}(n)$.

Lösungsvorschlag zu Aufgabe 2

a) Folgt sofort mit

$$\binom{n}{k} = \frac{n \cdots (n-k+1)}{k!} = \frac{n^k + O(n^{k-1})}{k!}.$$

b) Wir haben

$$\binom{2n}{n} = \frac{(2n) \cdot (2n-1) \cdots (n+1)}{n \cdot (n-1) \cdots 1} = \frac{2n}{n} \cdots \frac{2n-i}{n-i} \cdots \frac{n+1}{1}.$$

Jeder Term des Produkts lässt sich durch 2 abschätzen, da $i \geq 0$ und

$$\frac{2n-i}{n-i} \geq 2 \iff 2n - i \geq 2n - 2i.$$

Somit gilt $\binom{2n}{n} \geq 2^n$ und damit $\binom{2n}{n} = \Omega(2^n)$.

c) Bemerke $n^n = 2^{n \log_2 n}$, da

$$\lim_{n\to\infty} 2^n - n \log n = \infty$$

folgt die Behauptung.

d) Ersetze $x \mapsto 1/y$, dann ist

$$\lim_{x\to 0} \frac{x \log x}{1/x} = \lim_{x\to 0} x^2 \log x = \lim_{y\to\infty} \frac{-\log y}{y^2} = 0$$

da $\log y$ viel langsamer wächst. Somit ist tatsächlich $x \log x = o(1/x)$ für $x \to 0$.

e) Richtig; wir prüfen

$$\liminf_{n\to\infty} \frac{|(-2)^n|}{n} = \liminf_{n\to\infty} \frac{2^n}{n} = \infty.$$

f) Falsch, denn $\liminf_{n\to\infty} 1 + (-1)^n = 0$.

Lösungsvorschlag zu Aufgabe 3

(a) Für die Anzahl der Knoten folgt $|V| = |\{v_{i,j} \mid 1 \leq i, j \leq n\}| = n^2$ direkt aus der Definition von $M_{n,n}$. Um die Anzahl der Kanten zu bestimmen, stellen wir fest, dass jeder innere Knoten $v_{i,j}$ (d.h. $2 \leq i, j \leq n-1$) Grad 4 hat, die vier Knoten in den Ecken Grad 2 haben und alle übrigen Knoten am Rand Grad 3 haben. Mit Satz 1.5 aus dem Skript folgt

$$|E| = \frac{1}{2}\Big(4 \cdot (n-2)^2 + 2 \cdot 4 + 3 \cdot 4 \cdot (n-2)\Big) = 2n^2 - 2n.$$

Die Aufgabe lässt sich auch anders durch Abzählen der Kanten lösen.

(b) Es lässt sich leicht einsehen, dass ein kürzester Pfad von a nach b über $2(n-1)$ Kanten verläuft. Wieviele Möglichkeiten gibt es, einen solchen Pfad auszuwählen? Wir stellen fest, dass wir (ausgehend von a) in jedem Schritt entweder nach unten oder nach rechts zum nächsten Nachbarn gehen. Insgesamt machen wir $n-1$ Schritte nach unten und $n-1$ Schritte nach rechts. Von den insgesamt $2(n-1)$ Schritten können wir also $n-1$ Schritte auswählen, die nach rechts gehen sollen. Ist diese Auswahl getroffen, so führen alle anderen Schritte nach unten. Dies ergibt total $\binom{2(n-1)}{n-1}$ viele Möglichkeiten, die Schritte nach rechts auszuwählen. Wir stellen fest, dass einerseits jede solche Auswahl einen anderen $a-b-$Pfad liefert, und dass wir andererseits jeden kürzesten $a-b-$Pfad mitgezählt haben. Somit gibt es total $\binom{2(n-1)}{n-1}$ viele kürzeste Pfade.

(c) Per Voraussetzung ist $G = (V, E)$ ein zusammenhängender Teilgraph von $M_{n,n}$. Somit existiert per Definition 1.9 ein $a-b-$Pfad, also existiert auch ein kürzester $a-b-$Pfad. Um diesen zu finden, führen wir auf G Breitensuche aus, ausgehend von a. Mit Satz 1.20 folgt, dass am Ende in $d[b]$ die Länge des kürzesten $a-b-$Pfades steht. Diesen Wert können wir auslesen und sind fertig. Gemäss Satz 1.20 ist dieser Algorithmus korrekt und benötigt Laufzeit $\mathcal{O}(|V| + |E|)$. G ist ein Teilgraph von $M_{n,n}$, somit folgt mit Teilaufgabe (b), dass $|V| = \mathcal{O}(n^2)$ und $|E| = \mathcal{O}(n^2)$, womit wir eine Laufzeit von $\mathcal{O}(n^2)$ erhalten.

Lösungsvorschlag zu Aufgabe 4

Wir beobachten zunächst, dass ein Digraph $D = (V, A)$ keine topologische Ordnung besitzen kann, wenn er Zyklen (d.h. gerichtete Kreise) enthält. Ausserdem beobachten wir, dass jeder zyklusfreie Digraph mindestens einen Knoten ohne ausgehende Kanten enthält. Wir nennen solche Knoten *Senken* und stellen folgende Behauptung auf.

Lemma. *Sei $D = (V, A)$ ein Digraph und $v \in V$ eine Senke in D. Eine topologische Ordnung von $D \setminus \{v\}$, $f : V \setminus \{v\} \to \{1, ..., |V| - 1\}$ bildet zusammen mit $f(v) := |V|$ eine topologische Ordnung von D.*

Beweis. Wir nehmen an, dass es eine Kante $(u, w) \in A$ gibt, so dass $f(u) > f(w)$ gilt. Weil f auf $V \setminus \{v\}$ eine topologische Ordnung bildet, muss $w = v$ sein. Dies führt aber zum Widerspruch, weil per Definition für alle $u \in V \setminus \{v\}$ gilt $f(u) < f(v) = |V|$. □

Mittels modifizierter Tiefensuche wollen wir nun eine topologische Ordnung zu finden. Wir stellen den Status jedes Knoten durch eine Farbe dar: Schwarze Knoten v wurde bereits abgearbeitet, d.h. ihnen wurde bereits ein Wert $f[v]$ zugewiesen. Graue Knoten sind gerade auf dem Stack der Tiefensuche, und weisse Knoten wurden noch nicht betrachtet. Falls wir während der Tiefensuche auf einen grauen Knoten stossen, so haben wir einen Zyklus entdeckt und geben 'Keine topologische Ordnung existiert' aus. Immer wenn alle Nachbarn eines Knotens v abgearbeitet wurden, ist v im grau-weissen Restgraphen eine Senke und wir ordnen $f[v]$ den grösst möglichen Wert zu.

Einige Bemerkungen zum Pseudocode:

- Die äussere **for**-Schleife garantiert, dass die Tiefensuche den ganzen Graphen abarbeitet und der Algorithmus nicht terminiert, bevor allen Knoten ein Wert $f[v]$ zugewiesen wurde.
- Wir gehen davon aus, dass jeder Knoten zwei Adjazenzlisten, diejenige der ausgehenden und diejenige der einkommenden Kanten, besitzt. Wir verwenden lediglich die Liste der ausgehenden Kanten (dies genügt für DFS). Wir können auf den nächsten Knoten der Adjazenzliste in konstanter Zeit zugreifen. Dies wird analog wie in DFS implementiert: Für jeden Knoten v wird ein Pointer p_v abgespeichert, der jeweils auf den Knoten von v's Adjazenzliste zeigt, welcher als letztes von v aus besucht wurde. Jedesmal wenn die DFS zu v zurück kehrt, wird p_v eins weitergeschoben, was uns den nächsten Nachbarn in konstanter Zeit liefert.
- Sind alle Nachbarn von v abgearbeitet (dies ist der Fall sobald wir am Ende der Adjazenzliste von v ankommen), so sind alle Nachbarn schwarz gefärbt (siehe Korrektheit). Das heisst v ist im grau-weissen Restgraphen eine Senke und kann schwarz gefärbt werden.

Zur Korrektheit: Wir begründen zuerst, dass wenn wir am Ende der Adjazenzliste von v angekommen sind, dass dann v im grau-weissen Restgraphen eine Senke ist. Angenommen wir betrachten einen weissen Nachbarn u von v. Dann setzt die DFS v aufs Stack und betrachtet u. Die einzige Möglichkeit, dass die DFS wieder den Knoten v betrachtet, ist wenn beim betrachten von u ein S.POP() ausgeführt wird, was impliziert, dass u kurz zuvor schwarz gefärbt wurde. Das heisst, wenn die ganze Adjazenzliste von v durchlaufen wurde und kein grauer Knoten gefunden wurde, dann sind alle ausgehenden Nachbarn von v schwarz gefärbt.

Es ist klar, dass wir im Lauf einer Tiefensuche früher oder später entweder auf eine Senke des weiss-grauen Restgraphen oder einen bereits besuchten (also grauen) Knoten stossen. Im ersten Fall können wir diesem Knoten wie beschrieben einen Wert zuweisen, im zweiten Fall haben wir einen Zyklus gefunden, d.h. einen Beweis, dass es keine topologische Sortierung von G gibt. Wenn der zweite Fall nie eintritt, sind die Knoten zum Schluss topologisch geordnet. Es ist einfach zu sehen, dass für alle Kanten (u, v) gelten muss, dass $f(u) \leq f(v)$. Falls $f(u) > f(v)$ dann würde das bedeuten, dass u vor v einen Wert zugewiesen bekommen hätte. Dies ist aber nicht möglich, weil dann v als Nachbar von u zu diesem Zeitpunkt schwarz gefärbt wäre, was zu einem Widerspruch führt.

Zur Laufzeit: Die äussere *for*-Schlaufe geht genau einmal durch alle Knoten. Die Laufzeit von TOPSORT ergibt sich daher aus derjenigen der Tiefensuche, ist also $\mathcal{O}(|V| + |E|)$.

Algorithm 1 TOPSORT(Graph $G = (V, E)$)

```
for all v ∈ V do
  COLOR[v] :=WHITE
end for
S := new Stack
TIME := |V|
for all w ∈ V do
  if COLOR[w] =WHITE then
    v := w
    repeat
      Sei u der nächste Knoten aus der Adjazenzliste der ausgehenden Kanten von v. Setze
      u = nil falls wir am Ende der Liste angelangt sind.
      if COLOR[u] =WHITE then
        COLOR[v] := GREY
        S.Push(v)
        v := u
      else if COLOR[u] =GREY then
        return "topological sort doesn't exist"
      else if u=nil then
        COLOR[v] := BLACK
        f[v] := TIME
        TIME := TIME − 1
        v := S.Pop()
      end if
    until v = nil
  end if
end for
Gib (v, f[v]) für alle v ∈ V aus
```

Lösungsvorschlag zu Aufgabe 5

Sei L die Menge der Blätter des Baums $T = (V, E)$. Es bezeichne $T[V \setminus L]$ den Graphen, der entsteht, wenn man von T die Blätter entfernt.

Lemma. *Die Zentren von T und $T[V \setminus L]$ sind für $|V| \geq 3$ identisch.*

Beweis. Für einen Knoten $v \in V \setminus L$ gilt: Jeder Knoten u mit $d(v, u) = ecc(v)$ ist ein Blatt, da es sonst einen Knoten u' mit $d(v, u') > d(v, u)$ gäbe. Daraus folgt, dass sich die Exzentrizität von v um mindestens 1 verringert, wenn alle Blätter von T entfernt werden. Andererseits ist klar, dass der Nachbar eines Blatts u mit $d(v, u) = ecc(v)$ immer noch in $T[V \setminus L]$ ist und sich deshalb die Exzentrizität von v um höchstens 1 verringert. Es gilt also für alle Knoten $v \in V \setminus L$

$$ecc_{T[V \setminus L]}(v) = ecc_T(v) - 1.$$

Es ist klar, dass für $|V| \geq 3$ die Zentren von T keine Blätter sind. Da die Zentren von T bzw. $T[V \setminus L]$ als Knoten minimaler Exzentrizität bezüglich T bzw. $T[V \setminus L]$ definiert ist und sich diese Grössen um genau 1 unterscheiden, sind die Zentren von $T[V \setminus L]$ genau die Zentren von T. □

Aus obigem Lemma finden wir folgendes iterative Vorgehen, um das Zentrum von T zu bestimmen: Wir entfernen in jeder Phase jeweils alle Blätter von T, d.h. sei T_i der Baum zu Beginn der i-ten Iteration, dann wählen wir $T_{i+1} = T_i[V(T_i) \setminus L(T_i)]$. Wir brechen die Iteration ab, sobald $|V(T_k)| \leq 2$ erreicht ist. $V(T_k)$ enthält dann genau die Zentren von T.

Die eleganteste Art, diese Idee in einen Algorithmus FINDCENTER (siehe Pseudocode) umzusetzen, ist, die Blätter in einer Queue zu führen. Wir nehmen an, dass T als Adjazenzlisten gegeben

Algorithm 2 FINDCENTER(Tree $T = (V, E)$)

```
Q := new Queue
for all v ∈ V do
  DEGREE[v] := deg v
  if DEGREE[v] = 1 then
    Q.Insert(v)
  end if
end for
while Q.NotEmpty do
  v := Q.Dequeue()
  for all u ∈ Γ(v) do
    DEGREE[u] := DEGREE[u] − 1
    if DEGREE[u] = 1 then
      Q.Insert(u)
    end if
  end for
end while
return v
```

ist und bezeichnen mit T' den schrittweise zurechtgestutzten Baum. Während des Algorithmus lassen wir die Adjazenzlisten von T unverändert (!), zählen aber für jeden Knoten $v \in V$ seinen Grad in T' in der Variable DEGREE$[v]$ mit. In jedem Durchlauf der **while**-Schleife wird der vorderste Knoten v der Queue gedanklich aus T' entfernt und die Grade seiner Nachbarn entsprechend aktualisiert. (Natürlich hat v als Blatt höchstens einen Nachbarn in T', aber da wir nur die Adjazenzlisten von T haben, müssen wir alle Nachbarn in T durchlaufen, obwohl uns nur einer davon wirklich interessiert!) Sobald ein Nachbar u von v ein Blatt in T' wird, wird er in die Queue eingefügt. Nach jeder Iteration sind also die inneren Knoten von T' genau die Knoten mit DEGREE$[v] \geq 2$ und die Blätter von T' genau die Knoten in der Queue. Man beachte, dass die Variable DEGREE$[v]$ im Allgemeinen ins Negative heruntergezählt wird - dies tut nichts zur Sache, da wir ja jeweils nur testen, ob sie genau 1 ist oder nicht, um v im richtigen Moment in die Queue einzufügen.

Zur Korrektheit: Man überlegt sich leicht, dass der Algorithmus zuerst alle Blätter von $T_0 = T$ entfernt, dann diejenigen von $T_1 = T[V \setminus L]$ usw., so dass der letzte Knoten in der Queue genau ein Zentrum ist.

Zur Laufzeit: Die **while**-Schleife wird $|V|$-mal ausgeführt. Die darin enthaltene **for all**-Anweisung wird *total* höchstens $\sum_{v \in V} \deg(v) = 2|E|$-mal ausgeführt. $2|E|$ ist auch der Zeitbedarf für die Initialisierung des Arrays DEGREE zu Beginn. Es ergibt sich eine Laufzeit von $\mathcal{O}(|V| + |E|)$. (Das ist im Wesentlichen das Argument, mit dem in der Vorlesung und im Skript die Laufzeit der Breitensuche begründet wird.)

Da für Bäume zudem $|E| = |V| - 1$ gilt, ist die Laufzeit von FINDCENTER $\mathcal{O}(|V|)$.

Lösungsvorschlag zu Aufgabe 6

(a) Da genau $n - i$ Elemente in A grösser sind als a_i, ist a_i das k-grösste Element aller $2n$ Zahlen genau dann wenn genau $k - (n - i) - 1$ Elemente in B grösser sind wie a_i. Diese Aussage können wir leicht testen: Sei $z := k - (n - i) - 1$. Wenn $z < 0$ oder $z > n$, geben wir 'NEIN' aus. Ansonsten testen wir ob $b_{n-z+1} \geq a_i \geq b_{n-z}$ gilt. Falls Ja geben wir 'JA' aus ansonsten 'NEIN'. Beachte, dass wir $b_0 = -\infty$ und $b_{n+1} = +\infty$ definieren um sicher zu gehen, dass nie auf nicht existierende Elemente im Array zugegriffen wird. Alternativ könnte man die Fälle $z = 0$ und $z = n$ seperat betrachten. Korrektheit folgt aus obiger Aussage und die Laufzeit

ist offensichtlich $O(1)$.

(b) Sei y das k-grösste Element von A und B zusammen. Wir nehmen zunächst an, dass $y \in A$. Sei $1 \leq i \leq n$ beliebig. Wie zuvor setzen wir $z = k - (n - i) - 1$ und unterscheiden folgende Fälle:

- Falls $z < 0$, so folgt $n - i = k - 1 - z > k - 1$. Dann sind mindestens $k - 1$ Elemente von A grösser als a_i, was wiederum $y \in [a_i, \ldots, a_n]$ impliziert.
- Falls $z > n$, so folgt $(n - i) + n = k - 1 - z + n < k - 1$. Dann sind insgesamt weniger als $k - 1$ Elemente grösser als a_i, und wir haben $y \in [a_1, \ldots, a_{i-1}]$.
- Falls $0 \leq z \leq n$ und $a_i < b_{n-z+1}$, so sind mindestens z Elemente in B grösser als a_i. Deshalb sind insgesamt mindestens $n - i + z = k - 1$ Elemente grösser als a_i und es folgt $y \in [a_i, \ldots, a_n]$.
- Gilt schliesslich $0 \leq z \leq n$ und $a_i > b_{n-z+1}$, so sind höchstens $z - 1$ Elemente in B grösser als a_i. Deshalb sind insgesamt höchstens $z - 1 + n - i = k - 2$ Elemente grösser als a_i, also haben wir $y \in [a_1, \ldots, a_{i-1}]$.

Mithilfe dieser Eigenschaft können wir die Binäre Suche modifizieren, so dass A nach y durchsucht wird (Siehe Pseudocode). Anschliessend kann B mit demselben Algorithmus durchsucht werden. In einem der beiden Arrays ist die k-grösste Zahl enthalten, also finden wir sie entweder bei ersten oder zweiten Aufruf der modifizierten Binären Suche.

Die Korrektheit folgt aus obigen Überlegungen.

Der Algorithmus wird einmal für A und einmal für B ausgeführt, was eine Gesamtlaufzeit von $O(\log n)$ ergibt (die Laufzeitanalyse ist analog zur Analyse der Binären Suche).

Algorithm 3 FIND-K-LARGEST-ELEMENT (A, B, ℓ, r, k)

Eingabe: A,B sortierte Arrays, ℓ, r und k
Ausgabe: k-grösstes Element x, falls $x \in A$

```
b_0 ← −∞; b_{n+1} ← ∞;
if ℓ = r then
  Teste mit Alg. aus Teilaufgabe (a), ob a_ℓ das k-grösste Element ist;
  if a_ℓ ist k-grösstes Element then
    return a_ℓ
  else
    return „k-grösstes Element nicht in A“
  end if
else
  m ← ⌈(ℓ+r)/2⌉;
  z ← k − 1 − (n − m);
  if z < 0 or (z ≤ n and a_m < b_{n−z+1}) then
    return FIND-K-LARGEST-ELEMENT(A, B, m, r, k)
  else
    return FIND-K-LARGEST-ELEMENT(A, B, ℓ, m − 1, k)
  end if
end if
```

(c) Wir testen für jedes $a \in A$, ob es ein $b \in B$ gibt mit $a + b = x$. Dafür rufen wir jeweils BINÄRE-SUCHE$(B, 1, n, x - a)$ auf. Offensichtlich werden wir so das gesuchte Paar finden, falls es existiert. Da die BINÄRESUCHE n mal aufgerufen wird, ergibt sich eine Laufzeit von $O(n \log n)$.

(d) Sei zunächst $\ell = 1$ und $k = n$. Wir betrachten jeweils das Paar (a_ℓ, b_k). Falls $a_\ell + b_k = x$ so haben wir das gesuchte Paar gefunden. Falls $a_\ell + b_k > x$ so verringern wir k um 1 und falls $a_\ell + b_k < x$ erhöhen wir ℓ um 1. Anschliessend iterieren wir diese Prozedur für die neuen

Werte von ℓ und k so lange bis wir entweder ein Paar gefunden haben oder bis $\ell > n$ oder $k < 1$.

Algorithm 4 FIND-PAIR (A, B, x)

Eingabe: A,B sortierte Arrays, x
Ausgabe: $a \in A$, $b \in B$ mit $a + b = x$, falls so ein Paar existiert

```
while ℓ ≤ n and k ≥ 1 do
  ℓ ← 1; k ← n;
  if a_ℓ + b_k > x then
    k ← k − 1;
  else if a_ℓ + b_k < x then
    ℓ ← ℓ + 1;
  else
    return (ℓ, k)
  end if
end while
return „das gesuchte Paar existiert nicht“
```

Korrektheitsbeweis: Der Algorithmus terminiert: Solange das Paar noch nicht gefunden wurde, wird in jedem Schritt der while-Schleife entweder k um 1 verringert oder ℓ um 1 erhöht. Falls keine a, b mit $a + b = x$ existieren, dann ist klar, dass der Algorithmus 'das gesuchte Paar existiert nicht' ausgibt. Zu zeigen bleibt, dass wir tatsächlich ein Paar ℓ', k' mit $a_{\ell'} + b_{k'} = x$ finden, falls es solche Paare gibt. Wir beoabchten, dass in jedem Schritt der while-Schleife entweder ℓ um 1 erhöht oder k um 1 verringert wird. Betrachte den ersten Zeitpunkt, wo wir auf ein k oder ein ℓ treffen, das in einem solchen gewünschten Paar k', ℓ' enthalten ist. Zu einem solchen Zeit haben wir also $\ell = \ell'$ oder $k = k'$. Nehmen wir an dass $\ell = \ell'$. Es folgt, dass zu diesem Zeitpunkt $k \geq k'$ gilt. Für $k > k'$ gilt $a_\ell + b_{k'} > a_\ell + b_k = x$. Deshalb wird k solange verringert, bis $k = k'$ gilt und somit das gesuchte Paar gefunden ist. Der Fall $k = k'$ funktioniert analog.

Laufzeitanalyse: Die while-Schleife wird maximal $2n - 1$ mal ausgeführt. Eine Iteration hat konstanten Zeitaufwand. Somit folgt, dass die Laufzeit $O(n)$ beträgt.

Lösungsvorschlag zu Aufgabe 7

Wir führen folgenden Algorithmus aus:

Algorithm 5 SUMX(L, x)

```
i, j of integer
i := 1
j := n
while i < j do
  if L[i] + L[j] < x then
    i := i + 1
  else if L[i] + L[j] > x then
    j := j − 1
  else
    return L[i], L[j]
  end if
end while
return “no solution”
```

Zur **Korrektheit** des Algorithmus:

Angenommen, es gibt $p < q$ so dass $L[p] + L[q] = x$. Wir haben zu zeigen, dass der Algorithmus das Paar $i = p,\ j = q$ tatsächlich findet. Das sieht man so ein: Da i (beginnend von $i = 1$) schrittweise erhöht, und j (beginnend von $j = n$) schrittweise erniedrigt wird, tritt irgendwann $i = p$ oder $j = q$ ein (da $p < q$). Sei also $i = p$ und $j > q$. Da $L[j] \geq L[q]$ gilt $L[i] + L[j] \geq L[p] + L[q] = x$, also wird der Algorithmus j so lange nach unten zählen, bis $L[i] + L[j] = x$ erreicht ist, was spätestens bei $j = q$ der Fall ist. (Falls L keine "echte" Menge ist, sondern Elemente mehrfach enthält, kann dies bereits früher passieren.) Tritt zuerst $j = q$ ein (mit $i < p$), argumentiert man analog.

Zur **Laufzeit** des Algorithmus:

Die **while**-Schleife von SUMX wird höchstens n-mal durchlaufen, da entweder i erhöht oder j erniedrigt wird. Die Operationen in jedem Schleifenduchlauf haben sicher Laufzeit $\mathcal{O}(1)$, was eine Laufzeit von $\mathcal{O}(n)$ für SUMX ergibt.

Lösungsvorschlag zu Aufgabe 8

a) Für jede Position kann eine von 4 Farben ausgewählt werden. Demnach gibt es $4^3 = 64$ verschiedene Tripel.

b) Sei A die Menge der Tripel die momentan noch in Frage kommen, und sei $x = |A|$ deren Anzahl. Bob rät als nächstes ein beliebiges Tripel b. Jedes Tripel in A hat 0, 1, 2 oder 3 Übereinstimmungen mit b. Dementsprechend unterteilen wir A in die vier Mengen A_0, A_1, A_2 und A_3. Beachte, dass A_3 maximal ein Element enthält, da nur ein Tripel an allen 3 Stellen mit b übereinstimmt. Nach dem Schubfachprinzip besitzt eine der anderen drei Mengen mindestens $\lceil \frac{x-1}{3} \rceil$ Elemente. Da wir das wahre Trippel a nicht kennen, liegt a im schlimmsten Fall in dieser Menge A_i mit mindestens $\lceil \frac{x-1}{3} \rceil$ Elementen (bzw. der sogenannte Gegenspieler würde a so auswählen, dass möglichst viele Tripel noch in Frage kommen, indem er ein a aus der grössten Menge A_i wählt). Dann erhält Bob i Übereinstimmungen und es kommen weiterhein $|A_i| \geq \lceil \frac{x-1}{3} \rceil$ Tripel in Frage.

c) Wenn Bob mit einer festen Strategie spielt, wählen wir a so, dass nach jedem Rateversuch von Bob noch möglichst viele Tripel möglich sind. Diese Anzahl reduziert sich in jeder Runde laut b) von x auf mindestens $\lceil \frac{x-1}{3} \rceil$. Demnach gibt es nach einer Runde noch mindestens 21, nach zwei Runden noch mindestens 7, nach drei Runden noch mindestens 2 mögliche Tripel. Somit hat Bob vor der vierten Runde noch minestens 2 Tripel als mögliche Antworten zur Auswahl und braucht somit im schlimmsten Fall eine fünfte Runde.

Lösungsvorschlag zu Aufgabe 9

Wir lösen die Aufgaben, indem wir den minimalen Spannbaum T von G so umbauen, dass ein minimaler Spannbaum T' von G' entsteht. Wir fassen G' auf als G mit einer modifizierten Gewichtsfunktion $w' : G \to \mathbb{N}$. Nach Annahme haben wir $w'(e) < w(e)$ für die modifizierte Kante e und $w'(f) = w(f)$ für alle anderen Kanten f.

Wenn man e zur Kantenmenge von T hinzufügt, erhält man genau einen Kreis C. Es bezeichne $e_{\max}$ die Kante aus C mit dem grösstem Gewicht unter w'. Dann ist $T' := T \cup \{e\} \setminus \{e_{\max}\}$ ein Spannbaum von G, da durch Entfernen einer Kante aus C der Graph wieder kreisfrei wird und zusammenhängend bleibt. (Es ist möglich, dass $e_{\max}$ gleich der modifizierten Kante e ist - dann ist $T' = T$.)

Lemma. *T' ist ein minimaler Spannbaum von G unter w'.*

Beweis. Sei $\widetilde{T}$ irgendein Spannbaum von G. Wir zeigen, dass $w'(\widetilde{T}) \geq w'(T')$.

- Wenn $\widetilde{T}$ die Kante e nicht enthält, gilt $w'(\widetilde{T}) = w(\widetilde{T}) \geq w(T) = w'(T) = w'(T') + w'(e_{\max}) - w'(e) \geq w'(T')$ wegen der Annahme, dass T minimaler Spannbaum von G unter w ist und nach Konstruktion von T'.
- Wenn $\widetilde{T}$ die Kante e enthält, entfernen wir diese und erhalten zwei getrennte Komponenten von $\widetilde{T}$. Eine der restlichen Kanten von C, nennen wir sie f, verbindet diese Komponenten und vervollständigt so einen neuen Spannbaum $\widetilde{T}_1$ mit Gewicht $w'(\widetilde{T}_1) = w'(\widetilde{T}) - w'(e) + w'(f)$. Wieder haben wir $w'(\widetilde{T}_1) \geq w'(T)$ wegen der Annahme, dass T minimaler Spannbaum von G unter w ist. Es folgt

$$w'(\widetilde{T}) = w'(\widetilde{T}_1) - w'(f) + w'(e) \geq w'(T) - w'(e_{\max}) + w'(e) = w'(T').$$

Algorithmisch lässt sich der Graph T' leicht aus T und e konstruieren: Mit Tiefensuche in $T \cup \{e\}$ findet man den Kreis C in Zeit $\mathcal{O}(|V|)$, und die schwerste Kante $e_{\max}$ in C findet man in Zeit $\mathcal{O}(|E(C)|) = \mathcal{O}(|V|)$.

Bemerkung: Die Tiefensuche hat einen Kreis gefunden, wenn sie ausgehend von einem Knoten v auf einen Nachbarn $u \in \Gamma(v)$ mit $pred[u] \neq nil$ stösst. Dann bilden $pred[v]$, $pred[pred[v]]$, $\ldots$, u den Kreis. Die Laufzeit dieses letzten Schrittes ist maximal $\mathcal{O}(|V|)$.

Es ist auch möglich mit Breitensuche dasselbe Resultat zu erhalten: Sobald man einen Nachbarn $u \in \Gamma(v)$ mit $pred[u] \neq nil$ findet, folgt man den beiden Pfaden $(u, pred[u], pred[pred[u]], \ldots)$ und $(v, pred[v], pred[pred[v]], \ldots)$ (parallel!), um den ersten gemeinsamen Knoten w zu finden. Dann ist $(u, pred[u], \ldots, w, \ldots, pred[v], v)$ ein Kreis.

Lösungsvorschlag zu Aufgabe 10

Es bezeichne $[i] = \{1, \ldots, i\}$. Es sei b_i die maximale Summe bis zum i-ten Element, welche das i-te Element enthält, das heisst, $b_i = \max_{k \in [i]} \sum_{j=k}^{i} a_j$.

Es gilt

$$\begin{aligned} b_i &= \max_{k \in [i]} \sum_{j=k}^{i} a_j \\ &= \max\{\max_{k \in [i-1]} \sum_{j=k}^{i} a_j, a_i\} \\ &= a_i + \max\{b_{i-1}, 0\} \end{aligned}$$

Mit Hilfe dieser Rekursionsformel erstellen wir folgendes dynamisches Program. Wir initialisieren $b_1 = a_1$ und berechnen b_i für $i = 2, \ldots, n$ mit der Formel. Anschliessend suchen wir den maximalen Wert b_i, dies gibt uns den Wert der maximalen Summe und gleichzeitig den Index k. Um auch den Index ℓ zu finden, setzen wir $j = k$ und solang $b_j \neq a_j$ verringern wir j um 1 (im Falle $b_j = a_j$ wurde der linke Rand ℓ der Summe b_k erreicht).

Korrektheit folgt aus der Korrektheit der Rekursionsformel.

Laufzeit: Die For-Schleife zur Berechnung der b_i benötigt $\mathcal{O}(n)$, das suchen des maximalen b_i auch $\mathcal{O}(n)$ und das finden von Index ℓ ebenfalls $\mathcal{O}(n)$. Daraus ergibt sich die gewünschte Laufzeit $\mathcal{O}(n)$.

Lösungsvorschlag zu Aufgabe 11

Wir wollen diese Aufgabe auf das (aus der Vorlesung bekannte) Rucksackproblem übertragen. Sei $B := \sum_{i=1}^{n} b_i$. Wir wollen die n Bücher so auf die Regale R_1 und R_2 verteilen, dass der Bücherstapel in beiden Regalen möglichst gleich gross sind. Im Idealfall wäre $\sum_{i \in R_1} b_i = \sum_{i \in R_2} b_i = B/2$, aber dies ist nicht immer möglich.

Wir nehmen ohne Beschränkung der Allgemeinheit an, dass am Ende $\sum_{i \in R_1} b_i \leq \sum_{i \in R_2} b_i$ gilt, also $\sum_{i \in R_1} b_i \leq B/2$. Das erste Regal soll nun unser Rucksack sein, und der Rucksack soll Kapazität $B/2$ besitzen. Wir stellen fest, dass wir diese Kapazität nicht überschreiten dürften, weil sonst das erste Regal zu stark beladen wäre.

Beim Rucksackproblem wollen wir innerhalb der gegebenen Kapazität einen möglichst grossen Profit erreichen. In unserer Aufgabe ist es das Ziel, dass die Kapazität so gut als möglich ausgelastet wird. Wir setzen also einfach $w_i := b_i$ und $p_i := b_i$ für jedes Buch i. Eine optimale Lösung maximiert dann $\sum_{i \in R_1} p_i = \sum_{i \in R_1} b_i$ unter der Bedingung $\sum_{i \in R_1} w_i = \sum_{i \in R_1} b_i \leq B/2$. Also entspricht eine optimale Lösung gerade einem optimalen Bücherstapel auf dem ersten Regal.

Wir wissen jetzt, dass wir die Aufgabe nichts anderes als ein Rucksackproblem ist. Wir benutzen Algorithmus 3.1 mit den oben definierten Gewichten, Profiten und Kapazitäten, um die optimale Lösung zu finden. Gemäss Satz 3.3 ist dieser Algorithmus dann korrekt und hat eine Laufzeit von $O(p_{max} \cdot n^2) = O(10n^2) = O(n^2)$.

Lösungsvorschlag zu Aufgabe 12

Sei $S = \langle s_1, \ldots, s_m \rangle$ und $T = \langle t_1, \ldots, t_n \rangle$.

a) Da $m = n$ und wir nur an Ersetzungen interessiert sind, genügt es alle Positionen i zu zählen, an denen $s_i \neq t_i$. Dies lässt sich offensichtlich in $\mathcal{O}(n)$ Zeit bewerkstelligen, siehe Algorithmus 6.

b) Wir überlegen uns zuerst eine rekursive Form der Levenshtein-Distanz. Schreibe $d(i, j)$ für die Distanz zwischen $\langle s_1, \ldots, s_i \rangle =: S_i$ und $\langle t_1, \ldots, t_j \rangle =: T_j$. Zuerst bemerken wir falls $s_i = t_j$ gilt, so gilt $d(i, j) = d(i-1, j-1)$ da eine Sequenz von Operationen die S_{i-1} in T_{j-1} umwandelt auch S_i in T_j umwandelt. Desweiteren beobachten wir, dass es nicht darauf ankommt in welcher Reihenfolge, die Elemente von S gelöscht, eingefügt oder geändert werden. Wir können annehmen, dass die Operation welche das letzte Element von S modifiziert als erstes ausgeführt wird. Falls also $s_i \neq t_j$ so muss entweder am Ende von S_i das Element t_j eingefügt werden, oder das Element s_i in t_j geändert werden oder das Element s_i gelöscht werden. Wir unterscheiden diese drei Fälle:

   Ändern Nach dem Ändern gilt $s_i = t_j$ und somit

   $$d_1(i, j) = d(i-1,\ j-1) + 1.$$

   Hinzufügen Wir fügen das Zeichen t_j zu S_i hinzu. Danach gilt $s_{i+1} = t_j$ und somit

   $$d_2(i, j) = d(i,\ j-1) + 1.$$

   Löschen s_i wird gelöscht und somit gilt

   $$d_3(i, j) = d(i-1,\ j) + 1.$$

   Somit ist die Rekursion

   $$d(i, j) = \begin{cases} d(i-1, j-1) & \text{falls } s_i = t_j, \\ \min\{d(i-1,\ j-1) + 1,\ d(i,\ j-1) + 1,\ d(i-1,\ j) + 1\} & \text{sonst.} \end{cases}$$

Dazu kommen die Randbedingungen

$$d(i,j) = \begin{cases} i & \text{falls } j = 0, \\ j & \text{falls } i = 0, \end{cases}$$

durch Löschen/Hinzufügen aller Zeichen in der jeweils nichtleeren Zeichenfolge.

Da $d(i,j)$ nur von Einträgen mit kleineren Indizes abhängt, können wir mit einem dynamischen Programm eine Tabelle $d[i,j]$ von oben links nach unten rechts ausfüllen, siehe Algorithmus 7. Die Levenstein-Distanz zwischen S und T ist dann gegeben durch $d(|S|,|T|)$. Die Laufzeit wird von den beiden Schleifen bestimmt und ist daher $\mathcal{O}(mn)$.

Algorithm 6 Hamming-Distanz

```
k := 0
for i = 1, ..., n do
  if s_i ≠ t_i then
    k := k + 1
  end if
end for
return k
```

Algorithm 7 Levenshtein-Distanz

```
for i = 0, ..., m do
  for j = 0, ..., n do
    if i = 0 then
      d[i,j] := j
    else if j = 0 then
      d[i,j] := i
    else
      if s_i = t_j then
        c := 0
      else
        c := 0
      end if
      d[i,j] := min{d[i-1, j-1] + c, d[i, j-1] + 1, d[i-1, j] + 1}
    end if
  end for
end for
```

Lösungsvorschlag zu Aufgabe 13

Gemäss Aufgabenstellung kann man die Firma als gewurzelten Baum $T = (V, E)$ darstellen (mit $n = |V|$), wobei die Wurzel w den Firmenchef bezeichnet. Als Erstes suchen wir die Wurzel w, und führen von w aus eine Breitensuche aus, umd für jeden Knoten i die Distanz $d(i,w)$ zu berechnen. Da der Graph ein Baum ist, geht das offensichtlich in Laufzeit $\mathcal{O}(n)$.

Für jeden Mitarbeiter i ist ein Geselligkeitswert $f(i)$ gegeben. Wir sagen, dass eine Auswahl $S \subset V$ zulässig ist genau dann wenn für kein $i \in S$ sein Vorgesetzter k (resp. sein Vaterknoten) in S enthalten ist. Für eine zulässige Auswahl S setzen wir $f(S) = \sum_{i \in S} f(i)$.

Nun wollen wir für jeden Knoten i die folgenden beiden Werte berechnen: $\alpha(i)$ sei der maximale Geselligkeitswert, den eine zulässige Auswahl innerhalb des Teilbaums von i erreichen kann. Ausserdem soll $\beta(i)$ der maximale Geselligkeitswert innerhalb des Teilbaums von i ohne i selbst sein.

Dazu starten wir mit den Knoten mit höchster Distanz $d(i, w)$ zur Wurzel, und gehen Level für Level hinauf bis wir bei der obersten Ebene (d.h. beim Firmenchef) angekommen sind. Wir bezeichnen mit $N(i)$ jeweils die Kinder eines Knotens i. Wir setzen

$$\beta(i) = \begin{cases} 0, & \text{falls } i \text{ ein Blatt ist,} \\ \sum_{j \in N(i)} \alpha(j), & \text{sonst,} \end{cases}$$

und

$$\alpha(i) = \begin{cases} \max\{0, f(i)\}, & \text{falls } i \text{ ein Blatt ist,} \\ \max\{\beta(i), f(i) + \sum_{j \in N(i)} \beta(j)\}, & \text{sonst.} \end{cases}$$

Mit diesen beiden Rekursionen können wir alle Werte $\alpha(i)$ und $\beta(i)$ von unten nach oben berechnen. Für jeden Knoten berechnen wir zwei Werte, und jede Kante $\{i, j\}$ betrachten wir zweimal (nämlich, wenn wir $\alpha(i)$ und $\beta(i)$ für den Vorgesetzten i von j berechnen). Also können wir alle Werte in Gesamt-Laufzeit $\mathcal{O}(n)$ berechnen. Wir behaupten, dass am Schluss in $\alpha(w)$ der maximalen Wert $f(S)$ einer zulässigen Auswahl S steht.

Wir zeigen diese Behauptung per Induktion über die Knoten. Ist i ein Blatt, so besteht der Teilbaum von i nur aus i selbst und deshalb ist $\beta(i) = 0$ und $\alpha(i) = \max\{0, f(i)\}$. Wollen wir für einen anderen Knoten i den Wert $\beta(i)$ berechnen, können aber aus den Teilbäumen von allen Kindern von i jeweils die beste Auswahl nehmen. Da wir hier i selbst nicht nehmen werden, dürfen in der Tat alle Kinder von i verwendet werden. Somit folgt $\beta(i) = \sum_{j \in N(i)} \alpha(j)$. Da wir alle Werte von unten nach oben berechnen, haben wir bereits alle Werte $\alpha(j)$ bestimmt und können darauf zugreifen.

Wollen wir schliesslich für ein Nicht-Blatt i den Wert $\alpha(i)$ berechnen, haben wir zwei Möglichkeiten: entweder ist i selbst in der besten Auswahl enthalten, oder nicht. Im ersten Fall zählen wir $f(i)$ mit dazu, dürfen aber jetzt keine Kinder von i verwenden. Darum summieren wir über alle $\beta(j)$ für $j \in N(i)$, den darin haben wir jeweils den maximalen Wert der Teilbäume berechnet, wenn j selbst nicht verwendet werden darf. Im zweiten Fall ist der resultierende Wert wieder $\beta(i)$. Dies zeigt die Korrektheit unserer Rekursionsgleichung. Insbesondere folgt, dass in $\alpha(w)$ tatsächlich der Wert $f(S)$ der besten Auswahl S steht.

Jetzt müssen wir noch die optimale Auswahl S selbst ausgeben. Dies tun wir mit Backtracking von oben nach unten: Wenn $\alpha(w) = \beta(w)$, so können wir den optimalen Wert ohne w selbst erreichen, wenn $\alpha(w) > \beta(w)$, so ist w selbst in der optimalen Auswahl vorhanden und wir fügen w zu S dazu. Im ersten Fall können wir mit den Kindern von w fortfahren und dort denselben Vergleich machen. Im zweiten Fall haben wir w ausgewählt. Dann dürfen wir die Kinder von w nicht zur Auswahl dazunehmen, und fahren stattdessen mit den Enkelkinder von w fort. So können wir den ganzen Baum von oben nach unten durchgehen, bis wir die optimale Auswahl S gefunden haben. Da wir jeden Knoten und jede Kante einmal durchgehen, hat das Backtracking ebenfalls lineare Laufzeit $\mathcal{O}(n)$.

Lösungsvorschlag zu Aufgabe 14

a) Da zwischen den Arrays keine Relation besteht, müssen wir jedes davon mit binärer Suche durchsuchen. Der Aufwand für die Suche in A_i ist $\log|A_i| = \log 2^i = \mathcal{O}(i)$, somit ist der Gesamtaufwand

$$\sum_{i=0}^{k} \mathcal{O}(i) = \mathcal{O}\left(k^2\right) = \mathcal{O}\left(\log^2 n\right).$$

b) Sei x das einzufügende Element. Angenommen n_j sei die "erste" Null in der Binärschreibweise von n, d.h.

$$j := \min\{i \mid n_i = 0\}.$$

Wir sortieren dann die Arrays $A_0, \ldots, A_{j-1}$ plus das neue Element x nach A_j. Der Aufwand für einen einzigen *merge pass* von zwei sortierten Arrays der Länge m ist höchstens $\mathcal{O}(m)$ (vgl. MERGESORT). Wir fügen zuerst A_0 und $\{x\}$ zu A'_0 zusammen, dann A'_0 und A_1 zu A'_1, usw. Im i-ten Schritt ist die Länge der beteiligten Arrays gerade $|A_i| = 2^i$. Wir erhalten eine Laufzeit von

$$\sum_{i=0}^{j} 2^{i+1} = 2^{j+2} - 1 = \mathcal{O}\left(2^j\right) \tag{1}$$

fürs Zusammenfügen.

Um die Laufzeit von n INSERT-Operationen abzuschätzen, bemerken wir dass nur $n/2$ der Schritte überhaupt ein Zusammenfügen benötigen, nur $n/4$ das Zusammenfügen von 2 Arrays, usw. Somit ist die totale Laufzeit – mit (1) und $k = \lfloor \log_2 n \rfloor$ – höchstens

$$\sum_{i=0}^{k} \frac{n}{2^i} \cdot \mathcal{O}\left(2^i\right) = n \sum_{i=0}^{k} \mathcal{O}(1) = \mathcal{O}(n \log n)\,.$$

Jede einzelne Operation braucht also nur $\mathcal{O}(\log n)$ amortisierte Zeit.

(Es ist uns nicht gelungen diese Abschätzung mit einer Potentialstrategie zu zeigen.)

Lösungsvorschlag zu Aufgabe 15

Die Operation CHANGE-KEY(H, x, k) soll den Schlüsselwert von Knoten x im Fibonacci-Heap H auf k ändern. Wir betrachten folgenden Algorithmus:

Algorithm 8 CHANGE-KEY(H, x, k)

DECREASE-KEY$(H, x, -\infty)$
EXTRACT-MIN(H)
key$[x] := k$
INSERT(H, x)

Der Algorithmus setzt zunächst den Schlüssel von x auf $-\infty$ (beziehungsweise kleiner als das derzeitige Minimum in H), und entfernt dann x mittels der EXTRACT-MIN Operation. Nun wird der Schlüssel von x geändert, und x wird als neuer Knoten wieder in H eingefügt.

Gemäss unserer Analyse von Fibonacci-Heaps hat der Befehl DECREASE-KEY$(H, x, -\infty)$ amortisierte Kosten $a(i) = \mathcal{O}(1)$ und der Befehl EXTRACT-MIN(H) hat amortisierte Kosten $a(i) = \mathcal{O}(\log n)$. Für die letzten beiden Befehle ist klar, dass sie Kosten $\mathcal{O}(1)$ haben, sowohl amortisiert als auch reell. Somit sind die amortisierten Kosten von unserem Algorithmus $\mathcal{O}(\log n)$, bezüglich dem gewählten Potential.

Lösungsvorschlag zu Aufgabe 16

(a) Wir werden der Einfachheit halber annehmen, dass $|V'| \geq 3$ ist. Für $|V'| \leq 2$ kann man einfach direkt bestimmen, ob ein Hamiltonkreis vorliegt, und einen entsprechenden Graphen ausgeben.

Sei $e = \{u, v\}$. Wir konstruieren $G = (V, E)$ mit $V = V' \cup \{a, b\}$ und $E = E' \cup \{\{u, a\}, \{v, b\}\}$.

Behauptung: Es gibt einen Hamiltonkreis in G', der e enthält $\Leftrightarrow$ Es gibt einen Pfad der Länge $|V| - 1$ in G.

Beweis: $\Rightarrow$: Wir nehmen an es gibt einen Hamiltonkreis in G', der e enthält. Wenn wir e aus diesem Hamiltonkreis löschen, erhalten wir einen Pfad der Länge $|V'| - 1$ mit den Endknoten u und v. Da G' ein induzierter Teilgraph von G ist, finden wir diesen Pfad auch

in G. Desweiteren enthält dieser Pfad alle Knoten von G ausser a und b. Wir können nun die Knoten a und b über die Kanten $\{u, a\}$ und $\{v, b\}$ mit dem Pfad verbinden und erhalten einen Pfad der Länge $|V| - 1$ in G.

$\Leftarrow$: Wir nehmen an es gibt einen Pfad P der Länge $|V| - 1$ in G. Dieser Pfad enthält somit alle Knoten in G, denn ein Pfad der Länge $k - 1$ enthält k verschiedene Knoten. Da a und b Grad 1 haben, müssen a und b die Endknoten von P sein. Wir löschen nun a und b und ihre inzidenten Kanten und erhalten somite einen Pfad P', welcher jeden Knoten in G' enthält. Da u und v die einzigen Nachbarn von a und b sind, sind u und v die Endknoten von P'. Weiter enthält der Pfad P' nur Kanten aus G', da die einzigen Kanten die nicht in G' sind, genau diejenigen sind, die wir gelöscht haben. Ausserdem ist die Kante e nicht im ursprünglichen Pfad P (und somit auch nicht in P') enthalten, da P sonst nur Länge 3 hätte, was der Annahme $|V'| \geq 3$ (woraus $|V| \geq 5$ folgt) widerspricht. Wir können also die Kante e zum Pfad P' hinzufügen und erhalten einen Hamiltonkreis in G', der e enthält.

(b) Wir betrachten den folgenden Algorithmus: Sei wieder $e = \{u, v\}$. Wie in Teilaufgabe (a) konstruieren wir $G = (V, E)$ mit $V = V' \cup \{a, b\}$ und $E = E' \cup \{\{u, a\}, \{v, b\}\}$. Nun rufen wir Bobs Algorithmus mit G als Eingabe auf. Die Ausgabe dieser Subroutine ist ein längster Pfad P in G. Nun berechnen wir die Länge von P. Wenn die Länge von P genau $|V| - 1$ ist, geben wir aus, dass G' einen Hamiltonkreis, der durch e geht, enthält, ansonsten geben wir aus, dass G' keinen solchen Hamiltonkreis enthält.

Die Korrektheit des Algorithmus folgt aus Teilaufgabe (a). Um die Laufzeit zu analysieren, betrachten wir die einzelnen Schritte. Das Hinzufügen von zwei Knoten und zwei Kanten kann mit der richtigen Datenstruktur in konstanter Zeit ausgeführt werden. Bobs Algorithmus mit G als Eingabe hat eine Laufzeit von $\mathcal{O}(|V|^2 + |E|^2)$. Da $|V| = |V'| + 2$ und $|E| = |E'| + 2$, ist diese Laufzeit in $\mathcal{O}(|V'|^2 + |E'|^2)$. Zuletzt können wir die Länge eines Pfades in Zeit $\mathcal{O}(|V|)$ berechnen. Die totale Laufzeit des Algorithmus wird also von der Laufzeit von Bobs Algorithmus dominiert und ist $\mathcal{O}(|V'|^2 + |E'|^2)$.

(c) Wir betrachten den folgenden Algorithmus: Als erstes suchen wir einen Knoten v mit kleinstem Grad in G'. Ist dieser Grad 0 oder 1, so wissen wir bereits, dass G' keinen Hamiltonkreis hat und geben eine negative Antwort aus. Ansonsten rufen wir für jede zu v inzidente Kante den Algorithmus aus (b) auf. Wenn wir für eine dieser Kanten einen Hamiltonkreis finden, der diese Kante enthält, geben wir eine positive Antwort aus, ansonsten eine negative.

Um die Korrektheit unseres Algorithmus zu zeigen, argumentieren wir zuerst, dass wir einen Hamiltonkreis finden, wenn einer existiert. Sei also H ein Hamiltonkreis in G'. In einem Kreis hat jeder Knoten Grad 2, also folgt aus der Existenz von H, dass jeder Knoten in G' mindestens Grad 2 haben muss. Der Algorithmus wird also nicht bereits nach dem ersten Schritt abbrechen. Weiter muss H durch v und damit auch durch zwei zu v inzidente Kanten gehen. Beim Aufruf des Algorithmus aus (b) für eine dieser Kanten finden wir also H (oder einen anderen Hamiltonkreis). Wenn G' hingegen keinen Hamiltonkreis enthält, wird auch der Algorithmus aus (b) in keinem Aufruf einen finden, und wir geben somit keine positive Antwort aus.

Für die Laufzeit betrachten wir wieder die einzelnen Schritte. Ein Knoten mit minimalem Grad kann in Zeit $\mathcal{O}(|E'|)$ gefunden werden, weil wir dafür jede Kante nur zweimal betrachten müssen. Den Algorithmus aus (b) rufen wir $deg(v)$ mal auf. Da $deg(v) \leq |V'| - 1$ folgt also eine totale Laufzeit von $\mathcal{O}(|E'| + |V'| \cdot (|V|^2 + |E'|^2)) = \mathcal{O}(|V'|^3 + |E'|^3)$.

(d) Sollte Bob tatsächlich recht haben, hätten wir einen effizienten Algorithmus gefunden, der entscheidet, ob ein Graph einen Hamiltonkreis besitzt. Dieses Problem ist jedoch NP-vollständig, ein solcher Algorithmus würde also $P = NP$ beweisen. Alice ist also zurecht skeptisch. Sie kann sich aber nicht absolut sicher sein, dass Bob falsch liegt, da es bisher auch keinen Beweis für $P \neq NP$ gibt.

Lösungsvorschlag zu Aufgabe 17

Um eine präzise Analyse durchzuführen, nehmen wir an dass die DNF-Formel die Form

$$F = D_1 \vee D_2 \vee \cdots \vee D_m, \qquad D_i = y_{i_1} \wedge \cdots \wedge y_{i_{k_i}},$$

hat, d.h. die i-te Klausel hat genau $k_i \leq k$ Literale. Insgesamt kommen also $K := \sum_i k_i$ Literale vor, was gerade die Grösse der Eingabe ist. Wir nehmen an, dass $n \leq K$ und dass die Variablen mit den Zahlen $\{1, \ldots, n\}$ nummeriert sind, da wir sonst mehr Variablen hätten als überhaupt in der Formel vorkommen können.

Es genügt, eine einzige erfüllbare Klausel zu finden. Man kann also die Klauseln D_i der Reihe nach durchgehen und für jede separat prüfen, ob es eine Belegung der Variablen gibt, die D_i erfüllt. Jede Klausel D_i besteht aus mit $\wedge$ verknüpften Literalen. Eine erfüllende Belegung der Klausel gibt es genau dann, wenn es keine Variable x gibt, die in D_i sowohl als positives Literal x als auch als negiertes Literal $\bar{x}$ auftritt.

Dies lässt sich in linearer Zeit überprüfen: Wir benutzen ein Array $s[j]$, das für jede Variable x_j (d.h. $j \in \{1, \ldots, n\}$) das "Vorzeichen" ihres Literals speichert. Wir gehen davon aus, dass Literale y_{i_j} als ganze Zahlen abgespeichert sind, deren Betrag die Variable angibt, und deren Vorzeichen angibt, ob die Variable negiert ist oder nicht (z.B. $y_{i_j} = -5$ heisst, dass das Literal $\overline{x}_5$ ist). Für jede Klausel D_i gehen wir durch die Literale y_{i_j} und notieren in $s[|y_{i_j}|]$ das Vorzeichen $\text{sign}(y_{i_j})$ des Literals. Wenn wir eine Variable finden, die bereits mit dem *entgegengesetzten* Vorzeichen aufgetreten ist, ist diese Klausel nicht erfüllbar ($p :=$ FALSE) und wir können direkt mit der nächsten Klausel fortfahren. Wenn wir keine solche Variable finden, haben wir eine erfüllbare Klausel gefunden und sind fertig. Diese "Hauptschleife" wird einmal pro Klausel ausgeführt und hat Laufzeit $\mathcal{O}(k_i)$. Um nicht vor jeder Klausel s komplett neu initialisieren zu müssen, gehen wir zur Vorbereitung bereits einmal die Literale der Klausel durch und setzen nur die Elemente von s auf 0, die tatsächlich in der Klausel vorkommen. Somit hat diese Initialisierung ebenfalls Laufzeit $\mathcal{O}(k_i)$ pro Klausel.

Insgesamt ist der Aufwand für die Schleife über alle Klauseln also $\sum_i \mathcal{O}(k_i) = \mathcal{O}(K)$. Angenommen wir benötigen $\mathcal{O}(n)$ Zeit um s zu reservieren, ergibt sich eine totale Laufzeit von $\mathcal{O}(K + n)$. Da aber $n \leq K$, ist dies wiederum $\mathcal{O}(K)$, also gerade linear in der Länge der Eingabe.

Algorithm 9 DNF-SAT

```
s := Array mit n Elementen
for i = 1, ..., m do
  s[|y_{i_j}|] := 0 für j = 1, ..., k_i
  p := TRUE
  for j = 1, ..., k_i do
    if s[|y_{i_j}|] ≠ 0 und s[|y_{i_j}|] ≠ sign(y_{i_j}) then
      p := FALSE
      break
    end if
    s[|y_{i_j}|] := sign(y_{i_j})
  end for
  if p then
    return "erfüllbar"
  end if
end for
return "nicht erfüllbar"
```

Lösungsvorschlag zu Aufgabe 18

Nehmen wir an, dass die Variable x_i k-mal in der 3-SAT Formel F vorkommt. Wir ersetzen das erste Vorkommen von x_i durch die neue Variable x_{i1}, das zweite Vorkommen durch x_{i2}, usw. Auf diese Weise ersetzen wir x_i durch k neue Variablen $x_{i1}, \ldots, x_{ik}$ und erhalten eine neue Formel F'. Wir müssen nun sicherstellen, dass in einer erfüllenden Belegung von F' diese k Variablen alle auf den selben Wahrheitswert gesetzt werden. Wir erzwingen dies, indem wir zu F' die Formel $H_i = (\neg x_{i1} \vee x_{i2}) \wedge (\neg x_{i2} \vee x_{i3}) \wedge \cdots \wedge (\neg x_{ik} \vee x_{i1})$ hinzufügen. Man beachte, dass H_i die Darstellung der Implikationskette $(x_{i1} \Rightarrow x_{i2}) \wedge (x_{i2} \Rightarrow x_{i3}) \wedge \cdots \wedge (x_{ik} \Rightarrow x_{i1})$ ist, d.h. H_i genau dann erfüllt ist, wenn alle x_{ij} den gleichen Wahrheitswert besitzen. Offensichtlich ist die so konstruierte Formel F'' genau dann erfüllbar, wenn F erfüllbar ist.

Dieses Verfahren wiederholen wir für jede Variable, die mehrfach in der ursprünglichen Formel F vorkommt. In der resultierenden Formel F'' kommen alle Variablen höchstens 3-mal vor, nämlich einmal in F', und zweimal in der jeweiligen Formel H_i. Ausserdem kommt jedes Literal höchstens 2-mal vor, da jede Variable einmal als nichtnegiertes und einmal als negiertes Literal in H_i auftritt.

Ausgehend von F lässt sich F'' leicht in polynomieller Zeit konstruieren – mit ein bisschen Nachdenken sogar in linearer Zeit. (Dies impliziert auch, dass die Länge von F'' höchstens polynomiell bzw. linear mit der Länge von F wächst.) Somit ist auch das restringierte Problem noch $\mathcal{NP}$-vollständig.