Technische Universität Darmstadt
Fachgebiet Theoretische Informatik

Prof. Johannes Buchmann
Nabil Alkeilani Alkadri
Nina Bindel
Patrick Struck

Algorithmen und Datenstrukturen

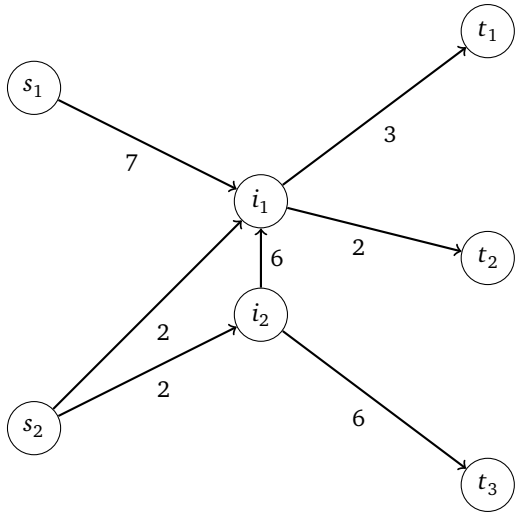
SoSe 2018

TECHNISCHE UNIVERSITÄT DARMSTADT

11. Lösungsblatt — 25.06.2018
v1.0

P1 Multiple Senken und multiple Quellen bei Maximalen Flüssen

Die Haushalte t_1, t_2, t_3 werden durch Wasser aus den Aufbereitungsanlagen s_1, s_2 durch folgendes Leitungssystem versorgt.



Hierbei beschreiben i_1, i_2 Verteilungszentren und die Kantengewichte geben den maximale Durchsatz (Kapazität) der jeweiligen Leitungsverbindung an.

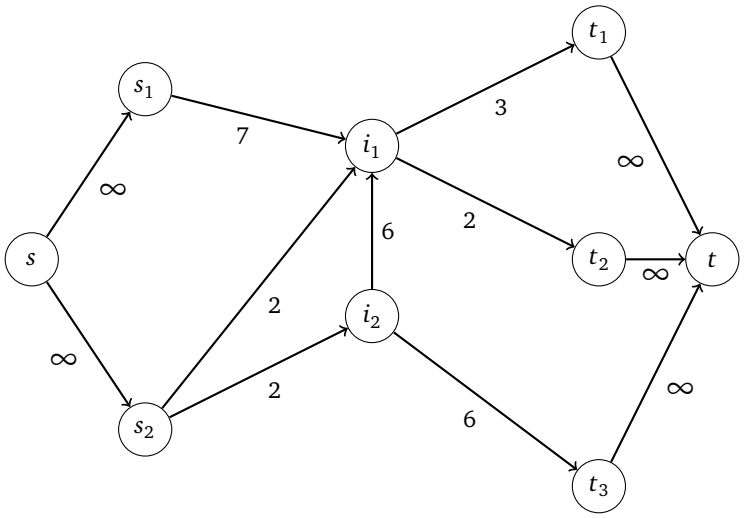
a) Erweitern Sie das Flussnetzwerk, sodass mithilfe der Basis-Version des Ford-Fulkerson Algorithmus der maximale Fluss in diesem Netzwerk bestimmt werden kann.

   Hinweis: Zusätzliche Kanten mit Kapazität ∞ verändern nicht die minimale Kapazität eines Pfades.

b) Wenden Sie den Ford-Fulkerson Algorithmus auf das in Aufgabenteil a) angepasste Flussnetzwerk an und bestimmen Sie den maximalen Fluss. Geben Sie außerdem nach jedem Iterationsschritt (jedem Schritt der **while**-Schleife) das Restnetzwerk an, einschließlich der Restkapazitäten. Skizzieren sie außerdem das abschließende Flussnetzwerk, wobei die Kanten mit dem Fluss und ihrer Kapazität in der Form $f(u,v)/c(u,v)$ gekennzeichnet sein sollen.

Lösung.

a) Um mehrere Quellen und mehrere Senken in das Flussnetzwerk zu integrieren, werden zwei weitere Knoten, eine Superquelle und eine Supersenke, hinzugefügt. Diese werden durch zusätzliche Kanten mit Kapazität $c(s,s_i) = \infty$ bzw. $c(t_i,t) = \infty$ mit allen tatsächlichen Quellen und Senken verbunden. Dies schränkt den Fluss eines Pfades von Quelle zu senken nicht ein, da ∞ als neutrales Element des Minimums die Bestimmungs der minimalen Restkapazität eines Pfades nicht beeinflusst.



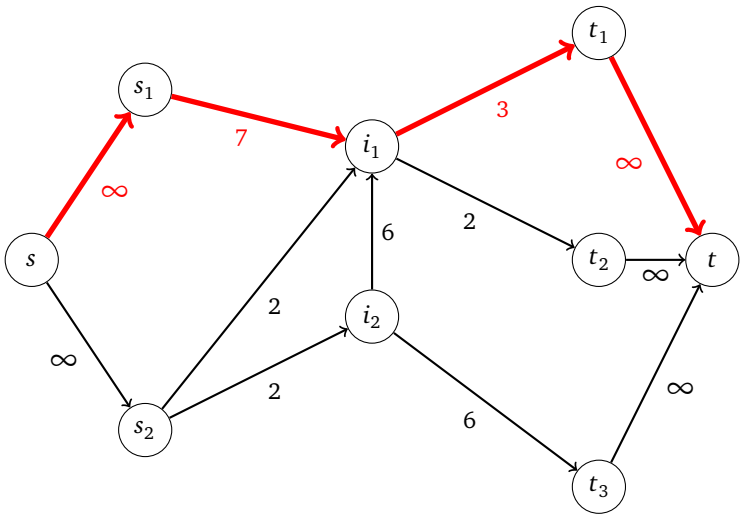
b) **Anmerkung:** Durch die freie Auswahl des Pfades beim Ford-Fulkerson Algorithmus können unterschiedliche Restnetzwerke entstehen.
Der maximale Fluss beträgt $|f| = 7$. Wir wählen folgende ergänzende Pfade:

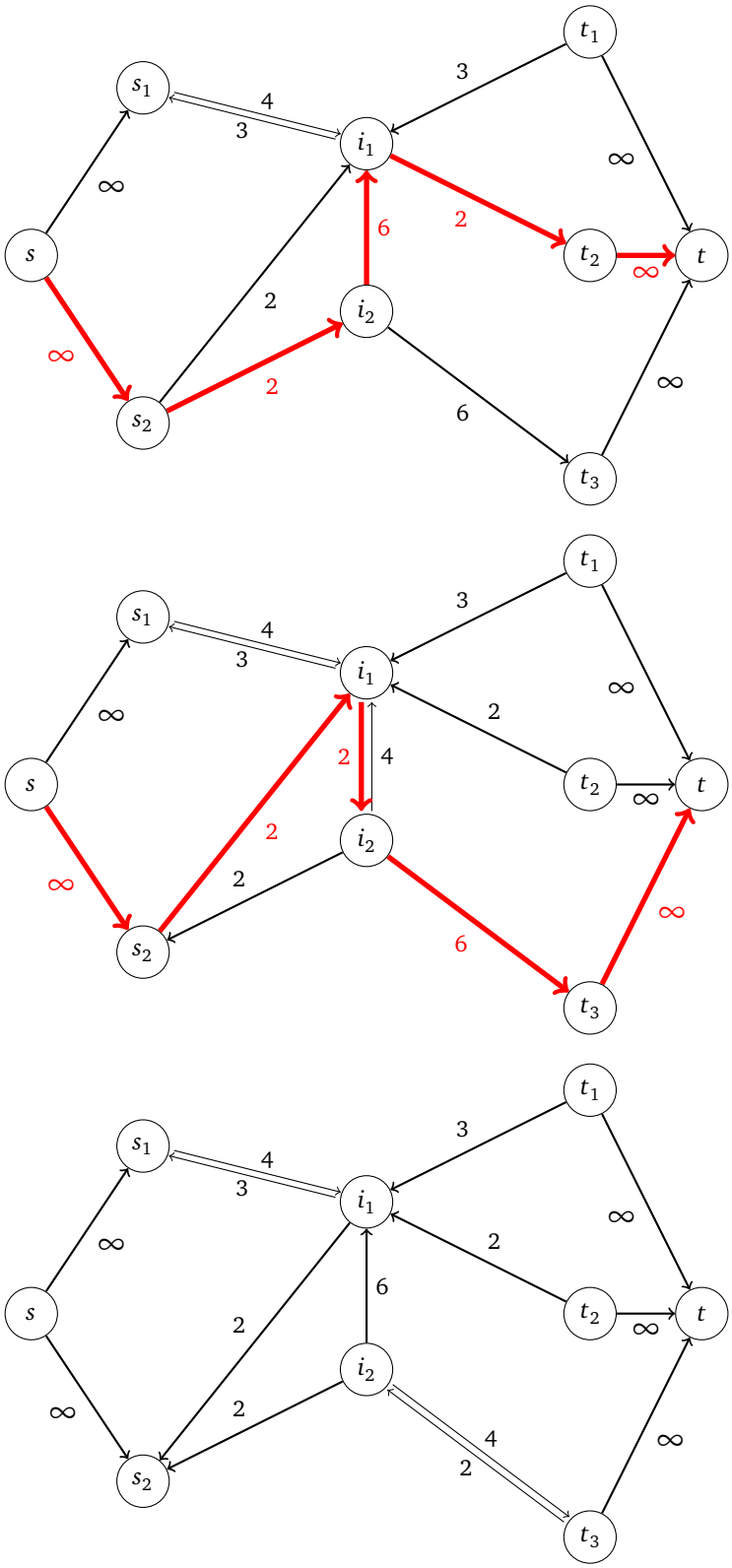
$$p_1 = \langle s, s_1, i_1, t_1, t \rangle \quad p_2 = \langle s, s_2, i_2, i_1, t_2, t \rangle \quad p_3 = \langle s, s_2, i_1, i_2, t_3, t \rangle$$

Zu diesen Pfaden folgen jeweils die maximalen Flüsse (als minimale Restkapazität auf einer der Kanten des Pfades):

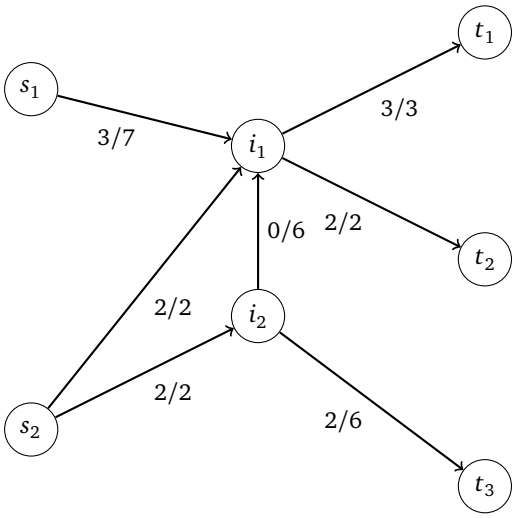
$$c_f(p_1) = 3 \quad c_f(p_2) = 2 \quad c_f(p_3) = 2$$

In den folgenden Grafiken werden die irrelevanten Kanten zur Quelle hin bzw. von der Senke her aus Gründen der Übersichtlichkeit nicht eingezeichnet.





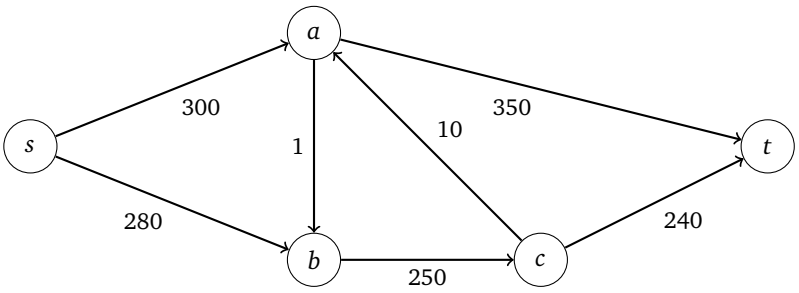
Das endgültige Flussnetzwerk:



P2 Edmonds-Karp Algorithmus

Der Edmonds-Karp Algorithmus modifiziert den Ford-Fulkerson Algorithmus, sodass statt einem beliebigen Pfad der kürzeste Pfad von s nach t im Restnetzwerk gewählt wird. Dies wird durch eine Breitensuche mit Startknoten s umgesetzt, bis ein Pfad nach t gefunden wurde (minimale Anzahl an Kanten durchlaufen).

a) Wenden Sie den Edmonds-Karp Algorithmus auf das nachfolgende Flussnetzwerk an und bestimmen Sie den maximalen Fluss. Um Zeit zu sparen, dürfen Sie darauf verzichten die Zwischenschritte zu zeichnen.



b) Erläutern Sie warum der Ford-Fulkerson Algorithmus bei obigem Netzwerk im Worst-Case deutlich mehr Pfade auswählen könnte, bis der maximale Fluss bestimmt ist. Beschreiben Sie die Art der ausgewählten Pfade im Worst-Case.

Lösung.

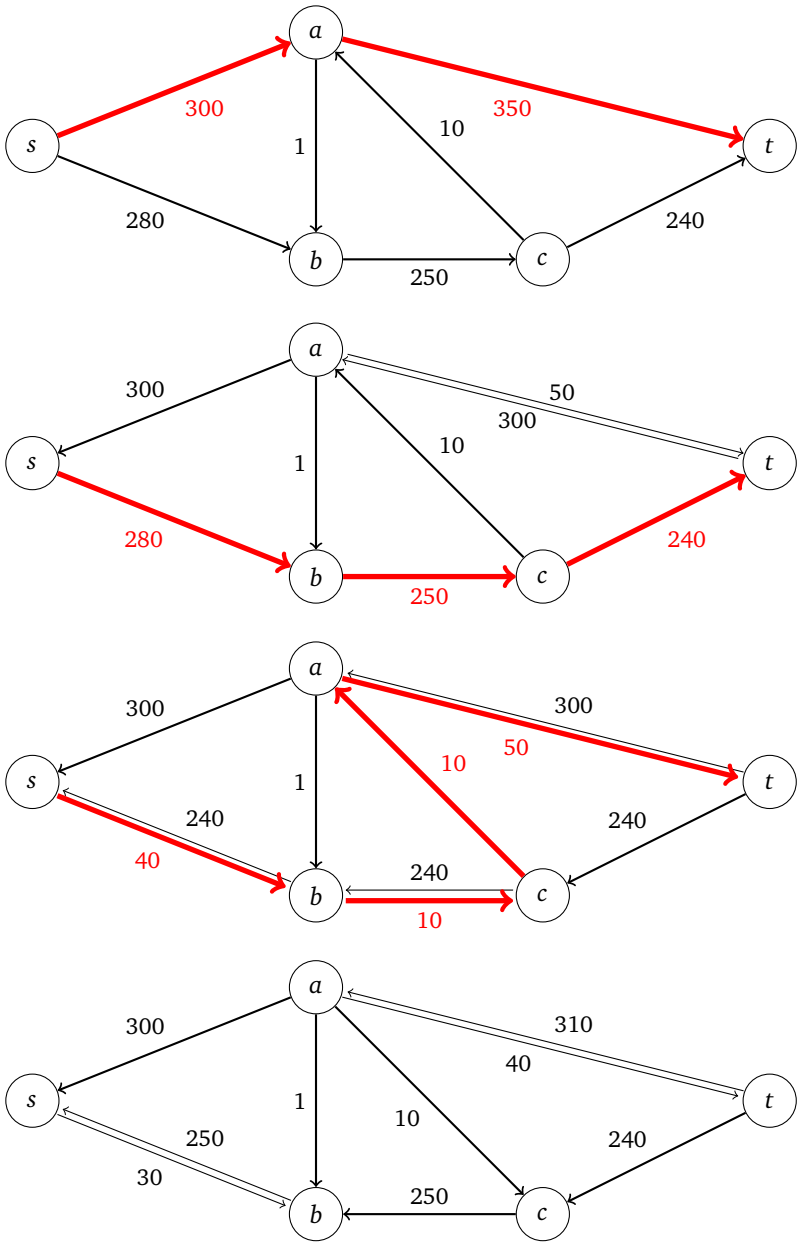
a) Der maximale Fluss beträgt $|f| = 550$. Wir wählen folgende ergänzende Pfade, die jeweils der kürzeste Pfad (Anzahl der Kanten) im aktuellen Restnetzwerk sind:

$$p_1 = \langle s, a, t \rangle \quad p_2 = \langle s, b, c, t \rangle \quad p_3 = \langle s, b, c, a, t \rangle$$

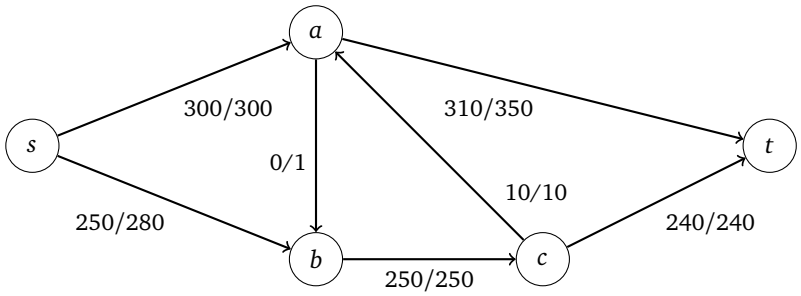
Zu diesen Pfaden folgen jeweils die maximalen Flüsse (als minimale Restkapazität auf einer der Kanten des Pfades):

$$c_f(p_1) = 300 \quad c_f(p_2) = 240 \quad c_f(p_3) = 10$$

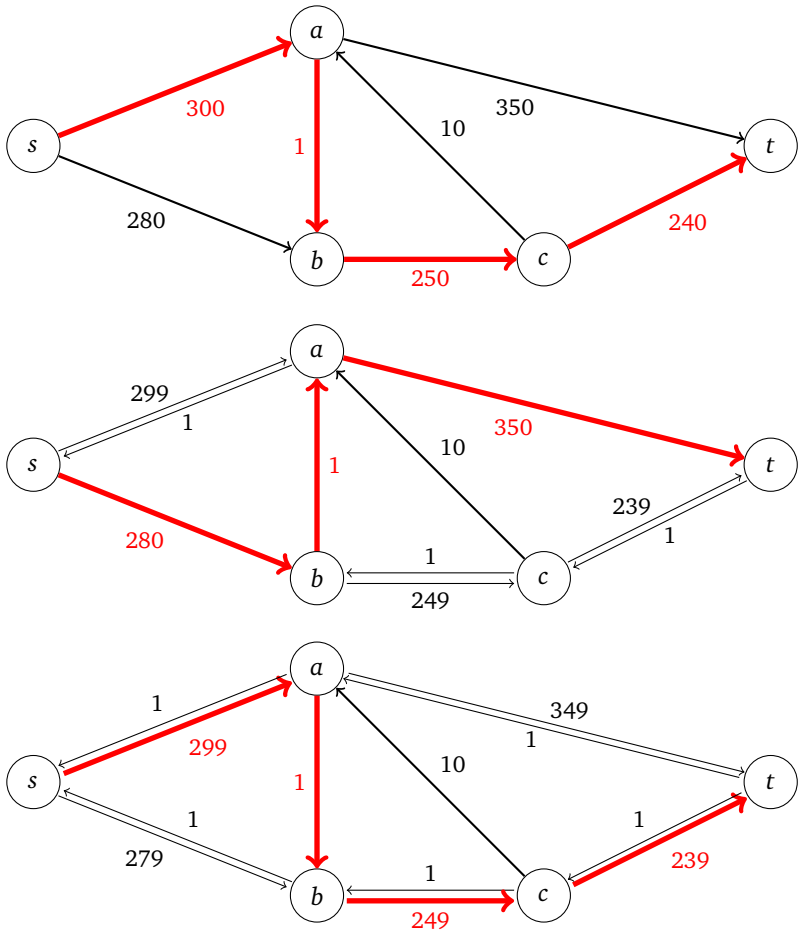
Im Folgenden werden die Restnetzwerke nach jeder Iteration eingezeichnet.



Das endgültige Flussnetzwerk:



b) Der Ford-Fulkerson Algorithmus wählt nicht zwangsläufig eine sinnvolle Reihenfolge der augmentierenden Pfade aus. Dies kann an diesem Beispiel illustiert werden, da der Ford-Fulkerson Algorithmus hier bis zu 482 Pfade auswählen könnte (Pfade mit Zyklen sind üblicherweise per Definition ausgeschlossen). In diesem Fall würden $2 \cdot 240$ Pfade ausgewählt werden, die den maximalen Fluss nur um eins erhöhen, wie in den folgenden Restnetzwerken dargestellt ist:



usw.

Außerdem würden noch die folgenden beiden Pfade ausgewählt werden:

$$p_{481} = \langle s, a, t \rangle \quad p_{482} = \langle s, b, c, a, t \rangle$$

Der Edmonds-Karp Algorithmus liegt mit dem in der Aufgabenstellung beschriebenen Auswahlverfahren in $O(VE^2)$ (vgl. Introduction to Algorithms), während der Ford-Fulkerson Algorithmus (mit natürlichen Zahlen als Kapazitäten) in $\Theta(E|f^*|)$ im Worst-Case liegt, da ähnlich zum verwendeten Beispiel, der Fluss möglicherweise pro Iteration nur um eins verbessert wird. Der Edmonds-Karp Algorithmus verhindert, dass ein Pfad mit einer Kante mit niedriger Restkapazität zu häufig ausgewählt wird.

P3 Dynamic Programming

Dynamic Programming ist eine Methode um die Laufzeit mancher Algorithmen zu verbessern, indem Zwischenergebnisse abgespeichert werden (time-memory trade-off).

a) Nennen Sie die zwei Bedingungen an Problemstellungen, damit die Anwendung von Dynamic Programming vorteilhaft sein kann.

b) Zwei Diebe haben Juwelen gestohlen und wollen ihre gemeinsame Beute nun fair aufteilen. Der Wert der n Objekte sei gegeben als $w_1, w_2, ..., w_n$ mit $w_i \leq k, \forall i, k$ mit $w_i \in \mathbb{N}$ und alle Werte verschieden, wobei k eine obere Schranke des Wertes eines Objektes darstellt.
Geben Sie einen Pseudocode-Algorithmus (in $O(kn^2)$ im Worst-Case) unter Verwendung von Dynamic Programming an, der entscheidet ob eine faire Aufteilung der Objekte möglich ist.

Hinweis: Ein naiver Algorithmus (in $O(2^n)$ im Worst-Case), der jede mögliche Aufteilung prüft, sei im Folgenden als Veranschaulichung angegeben. Dabei beinhaltet W die Werte der Objekte.

```
FAIR-PARTITION(W)
1  total = ∑_{i=1}^{n} W[i]
2  if total mod 2 ≠ 0
3      return false
4  return IS-SUBSET-POSSIBLE(W, W.length, total/2)

IS-SUBSET-POSSIBLE(W, n, sum)
1  if sum = 0
2      return true
3  if n = 0
4      return false
5  return IS-SUBSET-POSSIBLE(W, n-1, sum)
       ∨ IS-SUBSET-POSSIBLE(W, n-1, sum-W[n])
```

Sie können die nachfolgende Vorlage vervollständigen. Dabei können Sie annehmen, dass kein Wert größer als die Hälfte der Summe aller Werte ist und das mindestens ein Objekt existiert.
Vorgeschlagene Schleifeninvariante: Nach Iteration i ist partionable[j] genau dann **true**, wenn es eine Teilmenge von Werten des Teilarray W[1..i] gibt, deren Summe j ist.

```
FAIR-PARTITION-WITH-DP(W)
1  total = ∑_{i=1}^{n} W[i]
2  if total mod 2 ≠ 0
3      return false
4  let partitionable be a new boolean array from 0 to total/2
5  partitionable[0] = true
6  for i=1 to W.length
7
8
9
10
11
12
13
14
15
16
17
18
19
20
```

Lösung.

a) Die zwei Bedingungen lauten:

- **Optimal Substructure:** Die Lösung kann aus den Lösungen von Teilproblemen gewonnen werden. Beispielsweise wird die Fibonacci-Folge (für $n > 0$) mit $f_{n+2} := f_{n+1} + f_n$ berechnet.
- **Overlapping Subproblems:** Ein rekursiver Algorithmus bearbeitet das gleiche Teilproblem mehrmals. Beispielsweise folgt für die Fibonacci-Folge $f_{n+2} = f_{n+1} + f_n = 2 \cdot f_n + f_{n-1}$, wobei ein rekursiver Algorithmus ohne Memoisation f_n mehrmals unabhängig von vorherigen Ergebnissen berechnen würde.

b) Ein Algorithmus unter Verwendung von Dynamic Programming mit Bottom-Up in $O(kn^2)$:

```
FAIR-PARTITION-WITH-DP(W)
1  total = ∑_{i=1}^{n} W[i]
2  if total mod 2 ≠ 0
3      return false
4  let partitionable be a new boolean array from 0 to total/2
5  partitionable[0] = true
6  for i=1 to W.length
7      for j=total/2 decrementing to W[i]
8          partionable[j] = partitionable[j-W[i]] ∨ partionable[j]
9          if j=total/2 ∧ partitionable[j]
10             return true
11 return false
```

Anmerkung 1: Durch die Abhängigkeit von k kann dieser Algorithmus bei sehr hohen Werte ungeeignet sein. Möglicherweise können die Werte dann aber durch angemessene Normalisierung vorbereitet werden.

Anmerkung 2: Dieser Algorithmus kann derart erweitert werden, sodass er stattdessen die *bestmögliche* Aufteilung (bzw. die Teilsummen der bestmöglichen Aufteilung) bestimmt, was eher dem klassischen Optimierungsproblem entspricht, welche bei Dynamic Programming typisch sind. Beide Algorithmen basieren aber auf dem selben Grundprinzip, nämlich unter Verwendung von Speicher die Laufzeit zu minimieren.

P4 Top-Down Memoisation

Eine Folge sei rekursiv definiert über $a : \mathbb{N} \longrightarrow \mathbb{N} : i \longmapsto a_i$ mit:

$$a_n := a_{\lfloor n/2 \rfloor} + a_{\lfloor n/3 \rfloor} \quad mit \quad a_0 = a_1 = 1$$

Entwerfen Sie einen Top-Down-Algorithmus (rekursiv, mit zusätzlicher Initialfunktion) mit Memoisation zur Bestimmung des n-ten Wertes dieser Folge und bewerten Sie ob bei dieser Problemstellung ein Bottom-Up Algorithmus besser geeignet wäre.

Lösung.

```
MEMOIZED-SEQUENCE(n)
1 let A[0..n] be a new integer array
2 A[0]=A[1]=1
3 Mem-Seq-Aux(A,n)    //Memoized-Sequence-Auxiliary

MEM-SEQ-AUX(A,n)
1 if A[n] ≠ 0
2    return A[n]
3 else
4    A[n] = Mem-Seq-Aux(A, ⌊n/2⌋) + Mem-Seq-Aux(A, ⌊n/3⌋)
5    return A[n];
```

Bei diesem Algorithmus werden nur die Lösungen von einem Teil der Subprobleme benötigt, um das Ergebnis zu bestimmen (Z.b. sind bei Anwendung auf $n = 1000$ nur 31 Ergebnisse im Array eingetragen), sodass ein Top-Down Algorithmus, der nur die benötigten Werte berechnet, effizienter sein kann. Da aus der Bottom-Up-Sicht nicht trivial zu erkennen ist welche Subprobleme notwendig sind, wäre ein derartiger Algorithmus weniger geeignet.

H1 Maximale Flüsse - Anwendung

Geben sie zu folgenden Problemstellungen eine Beschreibung an, wie sie auf das Problem des Maximalen Flusses reduziert werden können, d.h. wie sie mit Anwendung des Ford-Fulkerson Methode gelöst werden können.

a) Gesucht ist die Anzahl unterschiedlicher Pfade im gerichteten Graphen $G = (V, E)$ von s nach t mit $s, t \in V$, wobei zwei Pfade unterschiedlich seien, wenn sie sich keine Kante teilen.

b) Einige Studierende, bezeichnet durch die Menge $S := \{s_1, s_2, ..., s_n\}$, möchten an einigen Projekten, bezeichnet durch die Menge $P := \{p_1, p_2, ..., p_m\}$, teilnehmen. Dabei bildet $maxS : S \longrightarrow \mathbb{N}$ einen Studierenden auf die Anzahl an Projekten ab, an denen er Interesse hat. Die Funktion $interest : S \longrightarrow \mathscr{P}(P)$ gibt die Projekte an, an denen ein Student Interesse hat und $maxP : P \longrightarrow \mathbb{N}$ wie viele Studierende maximal für ein Projekt zugelassen sind.
Gesucht ist die Anzahl von passenden Zuordnungen von Studierenden zu Projekten.

Wir betrachten zusätzlich ein Beispiel mit 3 Studenten s_1, s_2, s_3 und 3 Projekten p_1, p_2, p_3. Die Funktionenwerte seien wie folgt gegeben:

$$maxS(s_1) = 2, \quad maxS(s_2) = 1, \quad maxS(s_3) = 1$$
$$maxP(p_1) = 3, \quad maxP(p_2) = 2, \quad maxP(p_3) = 1$$
$$interest(s_1) = \{p_1, p_2\}, \quad interest(s_2) = \{p_2\}, \quad interest(s_3) = \{p_2, p_3\}$$

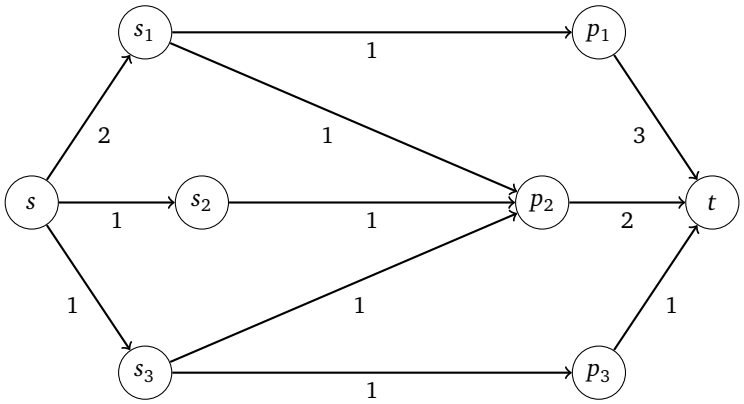
Geben Sie neben der allgemeinen Beschreibung, auch das zugehörige Netzwerk für das oben beschriebene Beispiel an.

Lösung.

a) Der Graph wird zu einem Flussnetzwerk erweitert, indem für jede Kante die Kapatizät 1 gesetzt wird. Nun entspricht der maximale Fluss dem gesuchten Wert.

b) Man erstelle das Flussnetzwerk (G, s, t, c) mit $G = (V, E)$, sodass $V = \{s, t\} \cup S \cup P$ und $E = \{(s, st) | st \in S\} \cup \{(pr, t) | st \in S\} \cup \{(st, pr) | st \in S \wedge pr \in interest(st)\}$ gelte. Die Kapazitäten von der Quelle aus sind durch $maxS$ und die Kapazitäten zu der Senke sind durch $maxP$ gegeben. Die Verbindungen zwischen Studierenden und Projekten haben Kapazität 1.

Für die Beispiel-Instanz ergibt sich folgendes Netzwerk:



H2 Dynamic Programming II

Entscheiden sie bei den folgenden Problemen, ob die Konzepte von Dynamic Programming hier angewendet werden kann, um das Zeitverhalten zu verbessern. Begründen sie kurz ihre Entscheidung zum Beispiel mit Verweis auf die in P3 a) behandelten Bedingungen oder durch Beschreibung eines Algorithmus.

a) Berechnung des Binomialkoeffizienten

b) Mergesort

c) In einem ungerichteten, zusammenhängenden Graphen: Bestimmung des Knotens, dessen Nachbarn den höchsten durchschnittlichen Grad haben.

d) Bestimmung jenes Teilarrays von einem Array ganzer Zahlen, sodass die Summe seiner Einträge maximal ist. Dabei sei ein Teilarray von einem Array bestimmt durch ein Start- und einen End-Index (i, j → A[i..j]), sodass er alle Werte A[k] beinhaltet, für die gilt $i \leq k \leq j$.
Beispielweise folgt bei Eingabe von [-3, 3, 4, -6, 3, 2] als Ergebnis [3, 4], da kein anderes Teilarray die Summe 7 übertrifft. Wäre die Eingabe stattdessen [-3, 3, 4, **-4**, 3, 2] folgt [3, 4, -4, 3, 2] mit der Summe 8.

Lösung.

a) Berechnung des Binomialkoeffizienten:
Ja. Zur Berechnung kann man über das pascalsche Dreieck und damit $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$ argumentieren, sodass wenn alle Binomialkoeffizienten mit kleinerem n und k zuvor in einem zwei-dimensionalen Array abgespeichert sind, das Ergebnis durch einfache Addition bestimmt werden kann (**optimal substructure**) und keine möglicherweise zeitintensivere Division und Multiplikation nötig ist. Die Eigenschaft **overlapping subproblems** folgt analog zur Fibonacci-Folge aus $\binom{n+2}{k+2} = \binom{n+1}{k+1} + \binom{n+1}{k+2} = (\binom{n}{k} + \binom{n}{k+1}) + (\binom{n}{k+1} + \binom{n}{k+2}) = \binom{n}{k} + 2 * \binom{n}{k+1} + \binom{n}{k+2}$.

b) Mergesort:
Mergesort ist ein Divide-and-Conquer Algorithmus, sodass keine überlappenden Teilprobleme vorhanden sind, da der Algorithmus die Liste in unabhängige Teillisten aufteilt, sie Lösung einer also keine Aufschluss auf die Lösung einer anderen gibt.

c) In einem ungerichteten, zusammenhängenden Graphen: Bestimmung des Knotens, dessen Nachbarn den höchsten durchschnittlichen Grad haben.
Dieses Problem besitzt die Eigenschaft **optimal substructure**, da aus dem Maximum zweier Teilgraphen, das Maximum des Gesamtgraphen folgt. **überlappende Teilprobleme** sind vorhanden, da der Grad eines Knotens von jedem Nachbarn aus abgefragt wird, um den eigenen Durchschnitt zu berechnen.

d) Das Teilarray eines Arrays ganzer Zahlen, sodass die Summe seiner Einträge maximal ist:
Ein naiver Algorithmus könnte schlicht alle Paare von Start- und Endindizes durchprobieren. Stattdessen kann ein Dynamic Programming Algorithmus umgesetzt werden, indem nach jeder Iteration i das beste Teilarray (definiert durch Start- und Endindex, sowie die Summe) abgespeichert wird, dass in Element i:

```
MAX-SUB-ARRAY(A)
1  (start, end, sum) = (1,0,0)           //The optimal subarray of A[0..i]
2  (start_c, end_c, sum_c) = (1,0,0) //The optimal subarray of A[0..i] with end_c = i
3  for i=1 to A.length
4     end_c = i
5     if sum_c > 0
6        sum_c = sum_c + A[i]
7     else
8        sum_c = A[i]
9        start_c = i
10    if sum_c > sum
11       (start, end, sum) = (start_c, end_c, sum_c)
12 return A[start..end]
```