Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Institut für Theoretische Informatik
Peter Widmayer
Tobias Pröger

Musterlösung zur Prüfung
Datenstrukturen und Algorithmen
D-INFK

24. Januar 2013

Name, Vorname: _____

Stud.-Nummer: _____

Ich bestätige mit meiner Unterschrift, dass ich diese Prüfung unter regulären Bedingungen ablegen konnte und dass ich die untenstehenden Hinweise gelesen und verstanden habe.

Unterschrift: _____

Hinweise:

- Ausser einem Wörterbuch dürfen Sie keine Hilfsmittel verwenden.
- Bitte schreiben Sie Ihre Studierenden-Nummer auf **jedes** Blatt.
- Melden Sie sich bitte **sofort**, wenn Sie sich während der Prüfung in irgendeiner Weise bei der Arbeit gestört fühlen.
- Bitte verwenden Sie für jede Aufgabe ein neues Blatt. Pro Aufgabe kann nur eine Lösung angegeben werden. Ungültige Lösungsversuche müssen klar durchgestrichen werden.
- Bitte schreiben Sie **lesbar** mit blauer oder schwarzer Tinte. Wir werden nur bewerten, was wir lesen können.
- Sie dürfen alle Algorithmen und Datenstrukturen aus der Vorlesung verwenden, ohne sie noch einmal zu beschreiben. Wenn Sie sie modifizieren, reicht es die Modifikationen zu beschreiben.
- Die Prüfung dauert 120 Minuten.

Viel Erfolg!

Stud.-Nummer: _____

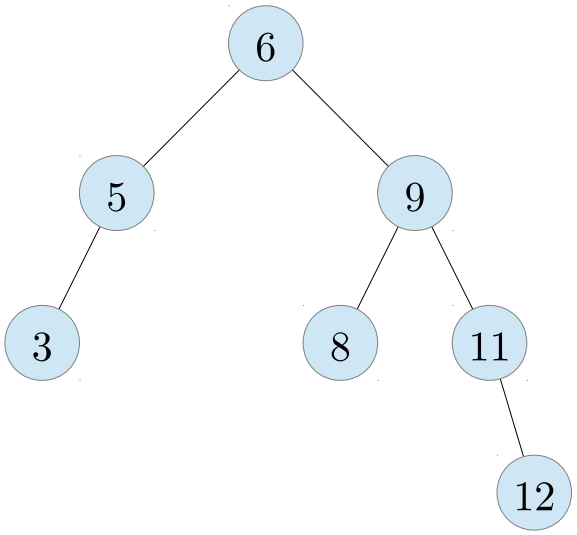
Aufgabe	1	2	3	4	5	**Σ**
Mögl. Punkte	8	7	9	9	10	43
$\sum$ Punkte						

Aufgabe 1.

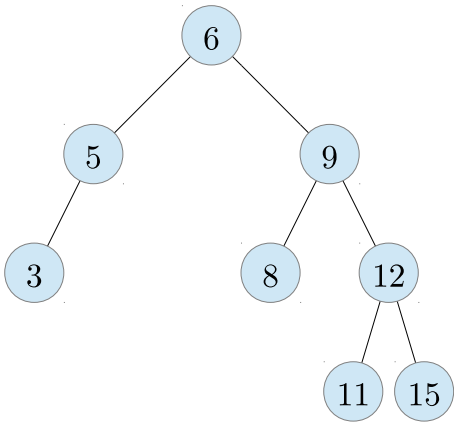
Hinweise:

1) In dieser Aufgabe sollen Sie **nur die Ergebnisse** angeben. Diese können Sie direkt bei den Aufgaben notieren.
2) Sofern Sie die Notationen, Algorithmen und Datenstrukturen aus der Vorlesung "Datenstrukturen & Algorithmen" verwenden, sind Erklärungen oder Begründungen nicht notwendig. Falls Sie jedoch andere Methoden benutzen, müssen Sie diese **kurz** soweit erklären, dass Ihre Ergebnisse verständlich und nachvollziehbar sind.
3) Als Ordnung verwenden wir für Buchstaben die alphabetische Reihenfolge, für Zahlen die aufsteigende Anordnung gemäss ihrer Grösse.

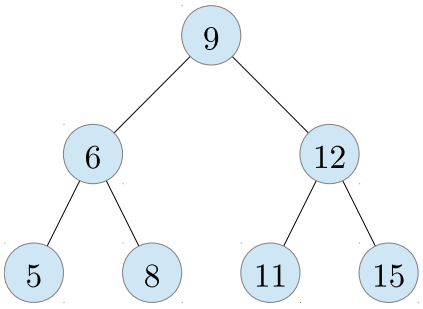
1 P (a) Fügen Sie in den untenstehenden AVL-Baum den Schlüssel 15 ein, und löschen Sie *danach* im entstandenen AVL-Baum den Schlüssel 3.



Nach Einfügen von 15:



Nach Löschen von 3:



1 P (b) Im untenstehenden Array sind die Elemente eines Max-Heaps in der üblichen Form gespeichert. Wie sieht das Array aus, nachdem das Maximum entfernt wurde und die Heap-Bedingung wieder hergestellt wurde?

35	33	22	20	24	10	16	12	11	17	14	9	5
1	2	3	4	5	6	7	8	9	10	11	12	13

33	24	22	20	17	10	16	12	11	5	14	9

1 P (c) Fügen Sie die Schlüssel 16, 6, 24, 41, 18, 38, 28 in dieser Reihenfolge in die untenstehende Hashtabelle ein. Benutzen Sie offenes Hashing mit der Hashfunktion $h(k) = k \bmod 11$ und lösen Sie Kollisionen mittels quadratischem Sondieren auf.

Bei quadratischem Sondieren kommen als Sondierungsfunktionen $s(j,k) = (-1)^j \cdot \lceil j/2 \rceil^2$ oder $s(j,k) = (-1)^{j+1} \cdot \lceil j/2 \rceil^2$ in Betracht. Die Position in der Hashtabelle werden durch $h(k) - s(j,k) \bmod 11$, $j \in \{0, 1, ..., 10\}$ beschrieben.

Falls $s(j,k) = (-1)^j \cdot \lceil j/2 \rceil^2$ verwendet wird:

$h(16) = 16 \bmod 11 = 5$: ok
$h(6) = 6 \bmod 11 = 6$: ok
$h(24) = 24 \bmod 11 = 2$: ok
$h(41) = 41 \bmod 11 = 8$: ok
$h(18) = 18 \bmod 11 = 7$: ok
$h(38) = 38 \bmod 11 = 5$: Kollision $\rightarrow h(38) + 1 = 6$: Kollision $\rightarrow h(38) - 1 = 4$: ok
$h(28) = 28 \bmod 11 = 6$: Kollision $\rightarrow h(28) + 1 = 7$: Kollision $\rightarrow h(28) - 1 = 5$: Kollision $\rightarrow h(28) + 4 = 10$: ok

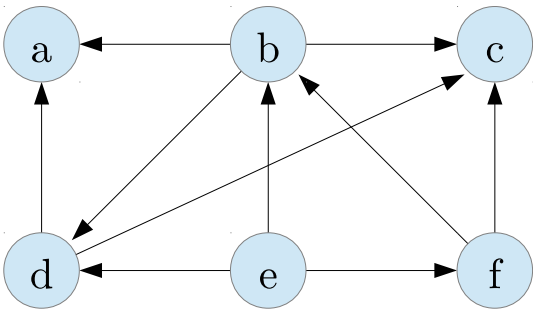
Falls $s(j,k) = (-1)^{j+1} \cdot \lceil j/2 \rceil^2$ verwendet wird:

$h(16) = 16 \bmod 11 = 5$: ok
$h(6) = 6 \bmod 11 = 6$: ok
$h(24) = 24 \bmod 11 = 2$: ok
$h(41) = 41 \bmod 11 = 8$: ok
$h(18) = 18 \bmod 11 = 7$: ok
$h(38) = 38 \bmod 11 = 5$: Kollision $\rightarrow h(38) - 1 = 4$: ok
$h(28) = 28 \bmod 11 = 6$: Kollision $\rightarrow h(28) - 1 = 5$: Kollision $\rightarrow h(28) + 1 = 7$: Kollision $\rightarrow h(28) - 4 = 2$: Kollision $\rightarrow h(28) + 4 = 10$: ok

In beiden Fällen sieht die Hashtabelle wie folgt aus:

		24		38	16	6	18	41		28
0	1	2	3	4	5	6	7	8	9	10

1 P (d) Geben Sie eine topologische Sortierung des untenstehenden Graphen an.

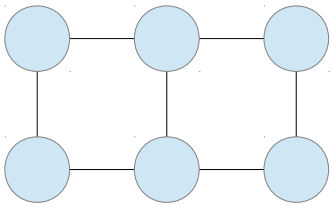


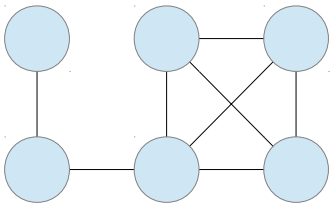
Dieser Graph besitzt zwei verschiedene topologische Sortierungen:

- e, f, b, d, a, c
- e, f, b, d, c, a

1 P (e) Geben Sie einen zusammenhängenden Graphen mit 6 Knoten an, der **genau** 3 verschiedene perfekte Matchings besitzt.

Hier sind verschiedene Lösungen möglich, z.B.





1 P (f) Wie viele Kanten kann ein bipartiter Graph mit n Knoten maximal besitzen? Sie dürfen der Einfachheit halber annehmen, dass n gerade ist.

Ein bipartiter Graph $G = (U \cup W, E)$ besitzt die maximale Anzahl Kanten, wenn jeder Knoten aus U mit jedem Knoten aus W verbunden wird. Ist $|U| = k$, dann ist $|W| = n - k$, und die Anzahl der Kanten beträgt

$$|E| = k \cdot (n - k).$$

Die maximale Anzahl von Kanten wird für $k = n/2$ erreicht und beträgt dann

$$n^2/4.$$

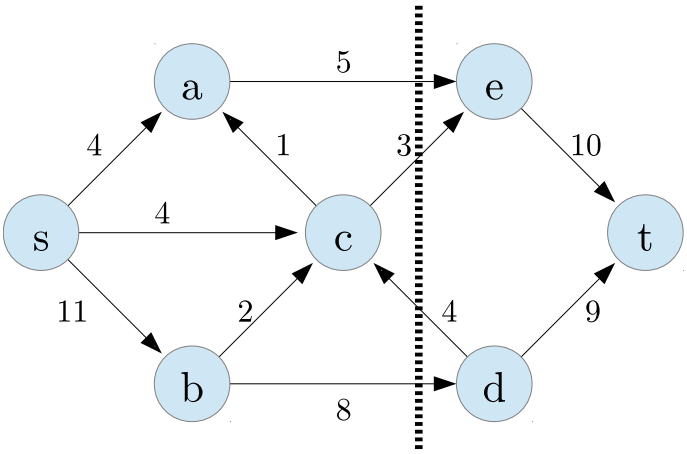
1 P (g) Das untenstehende Array wird mit *Sortieren durch Einfügen* sortiert. Geben Sie die an den ersten vier Vergleichen beteiligten Schlüsselpaare an.

8	15	6	12	4	9	10	11	13	19
1	2	3	4	5	6	7	8	9	10

Schlüsselpaare: (15, 8), (6, 15), (6, 8), (12, 15)

Alternative Lösung: (15, 8), (6, 8), (12, 6), (12, 8)

1 P (h) Gegeben ist das folgende Netzwerk mit Quelle s und Senke t. Die einzelnen Kapazitäten sind durch die Zahlen neben den Kanten gegeben.



Ein Fluss ϕ ist partiell durch die folgende Funktion gegeben:

(v, w)	(s, a)	(s, c)	(b, d)	(c, e)	(d, t)
$\phi(v, w)$	4	4	8	3	8

Ergänzen Sie die Flusswerte auf den übrigen Kanten, sodass ϕ ein gültiger Fluss ist. Dieser wird in jedem Fall maximal sein. Zeichnen Sie in die obige Abbildung einen Schnitt ein, der zeigt, dass ϕ maximal ist.

(v, w)	(s, b)	(a, e)	(b, c)	(c, a)	(d, c)	(e, t)
$\phi(v, w)$	8	5	0	1	0	8

Aufgabe 2.

1 P (a) Geben Sie für die untenstehenden Funktionen eine **Reihenfolge** an, so dass folgendes gilt: Wenn eine Funktion f links von einer Funktion g steht, dann gilt $f \in \mathcal{O}(g)$.

Beispiel: Die drei Funktionen n^3, n^7, n^9 sind bereits in der entsprechenden Reihenfolge, da $n^3 \in \mathcal{O}(n^7)$ und $n^7 \in \mathcal{O}(n^9)$ gilt.

- n^2
- $\binom{n}{4}$
- $\log(n^{17})$
- $(\log n)^5$
- $n!$
- $\sqrt{3^n}$
- 10^{50}

Lösung: Es gilt $\log(n^{17}) = 17\log(n) = \Theta(\log n)$. Damit ist $\log(n^{17}) = \mathcal{O}((\log n)^5)$. Weiterhin ist $\binom{n}{4} = \Theta(n^4)$, und $\sqrt{3^n} < n!$ für $n \geq 3$ und daher $\sqrt{3^n} = \mathcal{O}(n!)$.

Die einzige gültige Reihenfolge ist daher:

$$10^{50}, \log(n^{17}), (\log n)^5, n^2, \binom{n}{4}, \sqrt{3^n}, n!$$

3 P (b) Gegeben ist die folgende Rekursionsgleichung:

$$T(n) := \begin{cases} 8 + 5T(n/3) & n > 1 \\ 3 & n = 1 \end{cases}$$

Geben Sie eine geschlossene (d.h. nicht-rekursive) und möglichst einfache Formel für $T(n)$ an und beweisen Sie diese mit vollständiger Induktion.

Lösung: Da wir annehmen dürfen, dass n eine Potenz von 3 ist, gilt $n = 3^k$ für ein $n \in \mathbb{N}$. Wir teleskopieren, um auf eine Formel für $T(n)$ zu kommen:

$$\begin{aligned} T(n) &= 8 + 5T(n/3) \\ &= 8 + 5(8 + 5T(n/3^2)) = 8 + 5 \cdot 8 + 5^2 T(n/3^2) \\ &= 8 + 5 \cdot 8 + 5^2(8 + 5T(n/3^3)) = 8 + 5 \cdot 8 + 5^2 \cdot 8 + 5^3 T(n/3^3) \\ &= \ldots = 8 \sum_{i=0}^{k-1} 5^i + 5^k \cdot 3 = 8 \cdot \frac{5^k - 1}{5 - 1} + 5^k \cdot 3 = 2(5^k - 1) + 3 \cdot 5^k = 5^{k+1} - 2. \end{aligned}$$

Wir beweisen nun unsere Annahme durch vollständige Induktion über k.

Induktionsverankerung ($k = 0$): Es gilt $T(3^0) = T(1) = 3 = 5^{0+1} - 2$.

Induktionsannahme: Für ein $k \in \mathbb{N}_0$ sei $T(3^k) = 5^{k+1} - 2$.

Induktionsschritt ($k \to k + 1$):

$$T(3^{k+1}) = 8 + 5 \cdot T(3^k) \overset{\text{Ind.-Ann.}}{=} 8 + 5 \cdot (5^{k+1} - 2) = 8 + 5^{k+2} - 10 = 5^{k+2} - 2.$$

1 P (c) Geben Sie die asymptotische Laufzeit in Abhängigkeit von $n \in \mathbb{N}$ für den folgenden Algorithmus (so knapp wie möglich) in Θ-Notation an. Sie müssen Ihre Antwort nicht begründen.

```
1 for(int i = 0; i <= n; i ++) {
2         for(int j = 1; j <= n*n*n; j *= 2)
3                 ;
4         for(int k = 1; k <= n; k += 2)
5                 ;
6 }
```

Lösung: Die Schleife in Schritt 2 wird $\lceil \log_2(n^3) \rceil = \Theta(\log n)$ Mal durchlaufen. Die Schleife in Schritt 4 wird $\lceil n/2 \rceil = \Theta(n)$ Mal durchlaufen. Der Aufwand für die Schritte 2 – 5 beträgt also insgesamt $\Theta(n)$. Da sie $n+1$ Mal ausgeführt werden, ergibt sich eine Gesamtlaufzeit von $\Theta(n^2)$.

1 P (d) Geben Sie die asymptotische Laufzeit in Abhängigkeit von $n \in \mathbb{N}$ für den folgenden Algorithmus (so knapp wie möglich) in Θ-Notation an. Sie müssen Ihre Antwort nicht begründen.

```
1 for(int i = 1; i <= ⌈log_5(n)⌉; i ++) {
2         int k = n;
3         while(k > 1)
4                 k = k/4;
5 }
```

Lösung: Die Schleife in Schritt 3 wird genau $\lceil \log_4(n) \rceil = \Theta(\log n)$ Mal durchlaufen. Da die äussere Schleife genau $\lceil \log_5(n) \rceil + 1 = \Theta(\log n)$ Mal durchlaufen wird, ergibt sich eine Gesamtlaufzeit von $\Theta(\log^2(n))$.

1 P (e) Geben Sie die asymptotische Laufzeit in Abhängigkeit von $n \in \mathbb{N}$ für den folgenden Algorithmus (so knapp wie möglich) in Θ-Notation an. Sie müssen Ihre Antwort nicht begründen.

```
1 int f(int n) {
2         if(n == 1) return 1;
3         else {
4                 for(int i = 1; i <= n; i ++)
5                         ;
6                 return f(n/3)+1;
7         }
8 }
```

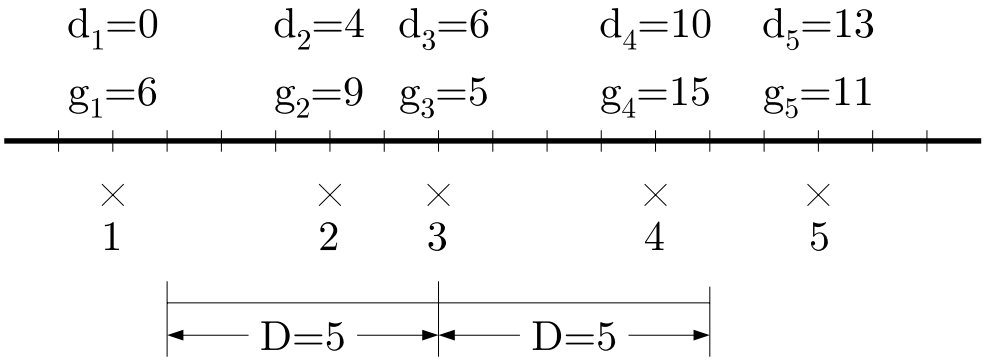
Lösung: Der Aufwand für die Schritte 1 – 8 ohne den rekursiven Aufruf in Schritt 6 ist linear in n, d.h. er beträgt maximal cn für eine geeignet gewählte Konstante $c > 0$. Die Gesamtlaufzeit (mit Berücksichtigung der rekursiven Aufrufe) ist dann durch

$$cn + c \cdot \frac{n}{3} + c \cdot \frac{n}{3^2} + ... = c\left(n + \frac{n}{3} + \frac{n}{3^2} + ...\right) < c\left(n + \frac{n}{2} + \frac{n}{2^2} + ...\right) \leq 2cn$$

nach oben beschränkt und beträgt damit ebenfalls $\Theta(n)$.

Aufgabe 3.

Entlang einer Strasse sollen Windräder zur Stromgewinnung aufgestellt werden. Aufgrund geographischer Gegebenheiten kommen n verschiedene Positionen in Betracht, aber Gesetze schreiben vor, dass zwischen zwei Windrädern ein Mindestabstand von D eingehalten wird. Die möglichen Positionen $d_1, ..., d_n$ sind als Koordinaten auf einer Linie angegeben, wobei die linkeste Position den Wert 0 hat. Anders ausgedrückt: Der Abstand der i-ten Position zur erstmöglichen Position beträgt d_i, es gilt $d_1 = 0$ und für alle $i \in \{1, ..., n-1\}$ ist $d_i < d_{i+1}$. Wenn ein Windrad an der Position i aufgestellt wird, dann beträgt der Gewinn $g_i > 0$. Die Aufgabe besteht nun darin, eine Positionierung von Windrädern mit maximalem Gewinn zu finden.



Beispiel: In der obigen Abbildung ist eine Situation für $n = 5$ mögliche Positionen dargestellt. Wird beispielsweise ein Windrad an der Position 3 aufgestellt, dann können an den Positionen 2 und 4 keine Windräder mehr aufgestellt werden. Werden die Windräder an den Positionen 1, 3 und 5 aufgestellt, dann beträgt der Gewinn $6 + 5 + 11 = 22$. Dies ist aber nicht die optimale Lösung: Eine Installation auf den Positionen 2 und 4 besitzt einen Gewinn von $9 + 15 = 24$.

1 P (a) Zeigen Sie anhand eines (möglichst einfachen) Beispiels, dass die folgende Greedy-Strategie nicht notwenigerweise eine optimale Lösung liefert: *"Wähle so lange aus den möglichen Positionen diejenige mit maximalem Gewinn aus, bis keine weiteren Windräder mehr platziert werden können."*

Lösung: Schon für $n = 3$ Windräder kann ein solches Beispiel konstruiert werden, z.B. mit den möglichen Positionen $d_1 = 0$, $d_2 = 1$, $d_3 = 2$ und einem Mindestabstand von $D = 2$. Weiterhin definieren wir die Gewinne als $g_1 = 2$, $g_2 = 3$ und $g_3 = 2$. Die beschriebene Greedy-Strategie platziert ein Windrad auf der Position 2. Da der jeweilige Abstand zu den Positionen 1 und 3 kleiner als D ist, können keine weiteren Windräder platziert werden und der Gewinn beträgt genau 3. Optimal wäre aber die Wahl der Positionen 1 und 3, die einen Gewinn von $2 + 2 = 4$ zur Folge haben.

5 P (b) Beschreiben Sie einen möglichst effizienten Algorithmus der *dynamischen Programmierung*, der den maximal erreichbaren Gewinn berechnet.

Lösung:

Definition der DP-Tabelle: Wir verwenden eine eindimensionale (!) Tabelle T mit n Einträgen. Der Eintrag $T[k]$ beschreibt den maximal erreichbaren Gewinn, wenn Windräder nur auf den Positionen $1, ..., k$ platziert werden können.

Berechnung eines Eintrags: Ist $d_k < D$ für eine Position $k \in \{1, ..., n\}$, dann wird der Mindestabstand zwischen der k-ten und der erstmöglichen Position nicht eingehalten, und es kann maximal eine Position aus den ersten k möglichen gewählt werden. Der maximale Gewinn wird dann erreicht, wenn eine Position aus $\{1, ..., k\}$ mit maximalem Gewinn gewählt wird. Wir setzen

$$T[1] := g_1 \text{ und } T[k] := \max\{T[k-1], g_k\} \text{ für alle } k > 1 \text{ mit } d_k < D,$$

und mit dieser Definition ist $T[k] = \max\{g_1, ..., g_k\}$ für alle k mit $d_k < D$.

Ist $d_k \geq D$, dann gibt es zwei Möglichkeiten:

- Es wird ein Windrad auf Position k platziert. Wenn wir

  $$v_k := \max\{i \in \{1, ..., k-1\} : d_i \leq d_k - D\}$$

  setzen, dann ist v_k der Index der Position links von k, die am nächsten an k liegt und den Mindestabstand D einhält. Auf den Positionen k' mit $v_k < k' < k$ können also *keine* Windräder platziert werden, da der Mindestabstand zu k nicht eingehalten wird.

  Der maximal erreichbare Gewinn mit den Positionen $1, ..., k$ besteht nun aus der Summe von g_k (Gewinn der Position k) und dem maximal erreichbaren Gewinn unter Verwendung von Positionen aus $\{1, ..., v_k\}$.

- Es wird kein Windrad auf Position k platziert. In diesem Fall ist der mit den Positionen $1, ..., k$ maximal erreichbare Gewinn genauso gross wie der maximal erreichbare Gewinn mit den Positionen $1, ..., k-1$.

Für alle k mit $d_k \geq D$ setzen wir also

$$\begin{aligned} v_k &:= \max\{i \in \{1, ..., k-1\} : d_i \leq d_k - D\} \\ T[k] &:= \max\{T[k-1], g_k + T[v_k]\}. \end{aligned}$$

Berechnungsreihenfolge: Da ein Tabelleneintrag nur von Einträgen mit kleinerem Index abhängt, können die Einträge $T[k]$ für aufsteigendes k berechnet werden. Analog können die Werte v_k für aufsteigendes k berechnet werden.

Auslesen der Lösung: Die Lösung steht am Ende im Eintrag $T[n]$.

1 P (c) Geben Sie die Laufzeit Ihres Algorithmus an.

Lösung: Eine sorgfältige Implementierung des obigen Verfahrens benötigt nur Zeit $\Theta(n)$, wenn v_k nicht in jedem Schritt naiv berechnet wird. Es ist klar, dass v_k nur für k mit $d_k \geq D$ berechnet werden muss. Für das erste solche k kann v_k wie in obiger Formel berechnet werden. Für alle weiteren k wissen wir, dass v_{k-1} bereits berechnet wurde, und es gilt

$$v_k = \max\{i \in \{1, ..., k-1\} : d_i \leq d_k - D\} = \max\{i \in \{v_{k-1}, ..., k-1\} : d_i \leq d_k - D\}.$$

Zur Berechnung von v_k aus v_{k-1} kann also initial $v_k = v_{k-1}$ gesetzt und dieser Index so lange um 1 erhöht werden, bis $d_{v_k+1} > d_k - D$ gilt. Auf diese Art können *alle* v_k in Zeit $\Theta(n)$ berechnet werden. Da die Laufzeit zur Berechnung eines einzelnen Eintrags $T[k]$ nur konstant ist, beträgt die Gesamtlaufzeit $\Theta(n)$.

2 P (d) Beschreiben Sie im Detail, wie der obige Algorithmus abgeändert werden muss, um eine Positionierung von Windrädern mit maximalem Gewinn zu berechnen.

Lösung: Wir füllen zunächst die gesamte Tabelle aus und speichern auch alle Werte von v_k. Eine optimale Positionierung von Windrädern kann nun durch Rückverfolgung der Werte in der Tabelle wie folgt erreicht werden. Wir setzen initial $k = n$ und wiederholen die folgenden Schritte solange $k \geq 1$ ist.

- Falls $d_k \geq D$ und $g_k + T[v_k] \geq T[k-1]$ ist, dann gib die Position k aus und setze $k := v_k$.
- Falls $d_k \geq D$ und $g_k + T[v_k] < T[k-1]$ ist, dann wird ein höherer Gewinn erzielt, wenn die Position k *nicht* benutzt wird. In diesem Fall wird nichts ausgegeben, und $k := k - 1$ gesetzt.
- Falls $d_k < D$ ist, dann kann nur ein einziges Windrad auf den Positionen $1, ..., k$ platziert werden. In diesem Fall wird die Position $i \in \{1, ..., k\}$ mit maximalem Gewinn g_i ausgegeben, und der Algorithmus terminiert.

Die Laufzeit dieses Verfahrens ist weiterhin $\Theta(n)$.

Aufgabe 4.

An einer Universität gibt es n Studenten, von denen jeder im nächsten Semester genau 5 Kurse besuchen muss. Insgesamt werden m Kurse angeboten, wobei am Kurs k maximal $T_k \in \mathbb{N}$ Studenten teilnehmen können.

Jeder Student i ist an einer Menge K_i von 10 Kursen interessiert. Die Universität möchte nun die Studenten so auf die Kurse verteilen, dass jeder Student nur Kurse besucht, die ihn interessieren, und dass kein Kurs k existiert, den mehr als T_k Studenten besuchen.

4 P (a) Es soll zunächst entschieden werden, ob eine solche Verteilung der Studenten auf die Kurse überhaupt existiert. Modellieren Sie dieses Problem als Flussproblem. Erläutern Sie genau, wie aus $K_1, ..., K_n$ und $T_1, ..., T_m$ ein geeignetes Netzwerk $N = (V, E)$ konstruiert werden kann. Geben Sie die Knoten V und die Kanten E der Konstruktion an und beschreiben Sie, welche Kapazitäten den Kanten zugewiesen werden müssen. Überlegen Sie, wie Sie aus dem Wert eines maximalen Flusses schlussfolgern können, ob eine geeignete Verteilung existiert oder nicht.

Lösung: Die Knotenmenge V besteht neben einer Quelle q und einer Senke s noch aus Knoten s_i für jeden Studenten $i \in \{1, ..., n\}$ und Knoten k_j für jeden Kurs $j \in \{1, ..., m\}$. Es werden die folgenden Kanten erzeugt:

- Für jeden Studenten $i \in \{1, ..., n\}$ wird die Kante (q, s_i) mit der Kapazität 5 erzeugt.
- Eine Kante (s_i, k_j) wird genau dann erzeugt, wenn der Student i sich für den Kurs j interessiert, d.h. wenn $j \in K_i$ gilt. Als Kapazität wird 1 festgelegt.
- Für jeden Kurs $j \in \{1, ..., m\}$ wird die Kante (k_j, s) mit der Kapazität T_j erzeugt.

Wir konstruieren also ein Netzwerk $N = (V, E, c)$ mit

$$\begin{aligned} V &:= \{q\} \cup \{s_i \mid i \in \{1, ..., n\}\} \cup \{k_j \mid j \in \{1, ..., m\}\} \cup \{s\} \\ E &:= \{(q, s_i) \mid i \in \{1, ..., n\}\} \cup \{(s_i, k_j) \mid i \in \{1, ..., n\}, j \in K_i\} \cup \\ &\qquad \{(k_j, s) \mid j \in \{1, ..., m\}\} \\ c((q, s_i)) &:= 5 \ \forall i \in \{1, ..., n\} \\ c((s_i, k_j)) &:= 1 \ \forall i \in \{1, ..., n\}, j \in K_i \\ c((k_j, s)) &:= T_j \ \forall j \in \{1, ..., m\}. \end{aligned}$$

Eine geeignete Verteilung der Studierenden existiert genau dann, wenn es im obigen Netzwerk N einen Fluss mit Wert genau $5n$ gibt.

2 P (b) Nennen Sie einen Flussalgorithmus, der das Problem aus (a) möglichst effizient löst. Nehmen Sie an, dass es mehr Studenten als Kurse gibt, und geben Sie die Laufzeit des genannten Verfahrens in Abhängigkeit von n und m an.

Lösung: Wir analysieren zunächst die Grösse des Netzwerks N in Abhängigkeit von n und m. Da es mehr Studenten als Kurse gibt, ist $m < n$. Neben einer Quelle und einer Senke gibt es n Knoten für die Studenten und m Knoten für die Kurse, also ist $|V| = 2 + m + n = \Theta(n)$. Jeder Knoten s_i, der einen Studenten repräsentiert, besitzt eine eingehende und 10 ausgehende Kanten. Für jeden Knoten k_j, der einen Kurs repräsentiert, kommt noch eine ausgehende

Kante hinzu, also ist $|E| = 11n + m = \Theta(n)$. Somit ist die Grösse des Netzwerks N nur linear in n (weshalb sich eine Adjazenzlistendarstellung zur Speicherung eignet). Ausserdem ist der Wert jedes Flusses in N durch $5n$ nach oben beschränkt.

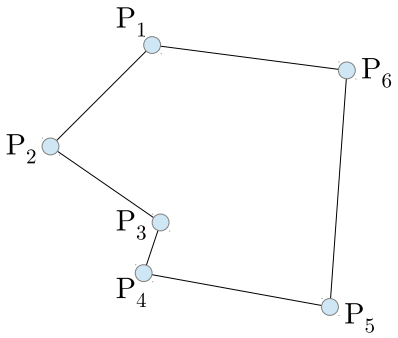
Es gibt nun verschiedene Flussalgorithmen zur Lösung. Beispielsweise berechnet der Algorithmus von Ford und Fulkerson einen maximalen Fluss mit $\mathcal{O}(|E| \cdot \phi^*)$ Operationen, wenn ϕ^* der maximale Flusswert ist. Damit ist seine Laufzeit pseudopolynomiell, und im Allgemeinen existieren effizientere Methoden zur Flussmaximierung. Da aber hier sowohl $|E| = \Theta(n)$ als auch $\phi^* = 5n = \Theta(n)$ gelten, beträgt die Laufzeit des Algorithmus von Ford und Fulkerson nur $\mathcal{O}(n^2)$, und die Wahl dieses Verfahrens ist *in diesem speziellen Fall* optimal.

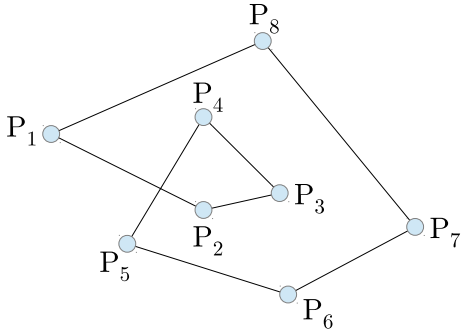
3 P (c) Wenn wir davon ausgehen, dass eine geeignete Verteilung der Studenten auf die Kurse existiert, dann interessieren wir uns natürlich auch dafür, wie diese aussieht. Beschreiben Sie einen Algorithmus, der für jeden Studenten i die Menge der 5 Kurse berechnet, die i im nächsten Semester belegen muss. Welche Laufzeit besitzt Ihr Verfahren, wenn ein maximaler Fluss im Netzwerk N bereits vorab berechnet wurde?

Lösung: Im Netzwerk N kann eine Breitensuche vom Knoten q gestartet werden, um die Knoten der Studenten s_i, $i \in \{1, ..., n\}$, zu finden. Von diesen verlaufen nur Kanten zu Kursen k_j, $j \in \{1, ..., m\}$. Für diese Kanten (s_i, k_j) kann der Flusswert $\phi((s_i, k_j))$ geprüft werden: Ist dieser 1, dann belegt der Student i den Kurs k_j, und das Paar (i, j) wird ausgegeben. Ansonsten ist der Wert 0 und der Student belegt den Kurs nicht. Auf diese Art werden alle Kurse von allen Studenten gefunden, und die Laufzeit beträgt $\Theta(|V| + |E|) = \Theta(n)$.

Aufgabe 5.

Es seien P_1, ..., P_n eine Menge von Punkten in $\mathbb{R}^2$. Wenn für alle $i \in \{1, ..., n-1\}$ jeweils die Punkte P_i mit P_{i+1} und zusätzlich P_n mit P_1 durch ein Liniensegment verbunden werden, dann entsteht ein *Polygon*.





4 P (a) Ein Polygon heisst *einfach*, wenn sich keine zwei Kanten eines Polygons kreuzen. In der obigen Abbildung ist nur das linke Polygon einfach. Das rechte Polygon ist nicht einfach, da sich die Segmente $\overline{P_1P_2}$ und $\overline{P_4P_5}$ kreuzen.

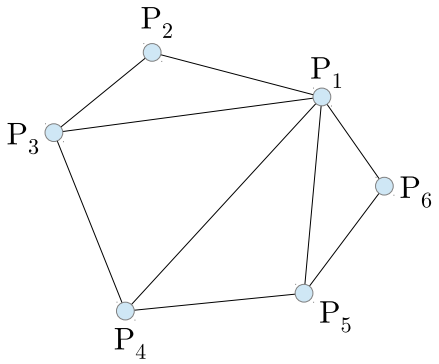
Geben Sie einen Algorithmus an, der nach dem Scanline-Prinzip arbeitet und in Zeit $\mathcal{O}(n \log n)$ testet, ob ein Polygon einfach ist.

Lösung: Das Problem kann auf das in der Vorlesung vorgestellte Schnittproblem für beliebig orientierte Liniensegmente reduziert werden. Das durch $P_1, ..., P_n$ definierte Polygon ist genau dann einfach, wenn sich die durch $\overline{P_1P_2}$, $\overline{P_2P_3}$, ..., $\overline{P_{n-1}P_n}$ und $\overline{P_nP_1}$ definierten Liniensegmente nicht schneiden. Der Test, ob es ein Paar sich schneidender Liniensegmente gibt, kann in Zeit $\mathcal{O}(n \log n)$ durchgeführt werden.

4 P (b) Wir nehmen nun an, dass P_1, ..., P_n ein konvexes Polygon definiert (Zur Erinnerung: Ein Polygon P heisst *konvex*, wenn die gesamte Verbindungsstrecke zwischen zwei beliebigen Punkten in P ebenfalls in P liegt).

Geben Sie einen möglichst effizienten Algorithmus an, der den Flächeninhalt des durch P_1, ..., P_n definierten konvexen Polygons berechnet. Welche Laufzeit besitzt Ihre Lösung?

Lösung: Wenn das Polygon konvex ist, dann sind die Punkte $P_1, ..., P_n$ bereits im oder gegen den Uhrzeigersinn geordnet und müssen nicht mehr sortiert werden.



Der Flächeninhalt des Polygons ist die Summe der Flächeninhalte der Dreiecke $\Delta P_1P_2P_3$,

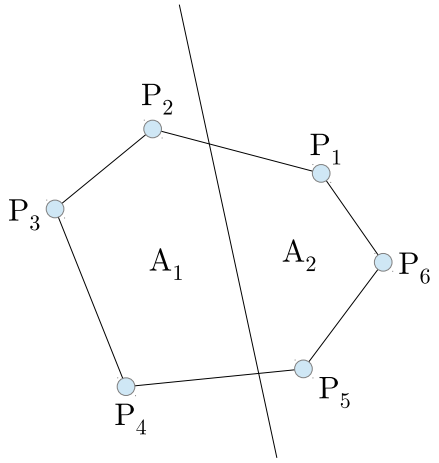
$\Delta P_1P_3P_4$, ..., $\Delta P_1P_{n-1}P_n$. Sind drei beliebige Punkte $P, Q, R \in \mathbb{R}^2$ gegeben, dann kann der Flächeninhalt des Dreiecks ΔPQR in Zeit $\mathcal{O}(1)$ bestimmt werden. Der folgende Pseudocode beschreibt den Algorithmus zur Berechnung des Flächeninhalts des durch $P_1, ..., P_n$ definierten Polygons:

ComputeArea$(P_1, ..., P_n)$

```
1 A ← 0
2 for i ← 2 to n − 1 do
3     Berechne den Flächeninhalt A_i des durch P_1, P_i und P_{i+1} definierten Dreiecks.
4     A ← A + A_i
5 return A
```

Da die Punkte $P_1, ..., P_n$ nicht sortiert werden müssen, beträgt die Laufzeit $\Theta(n)$.

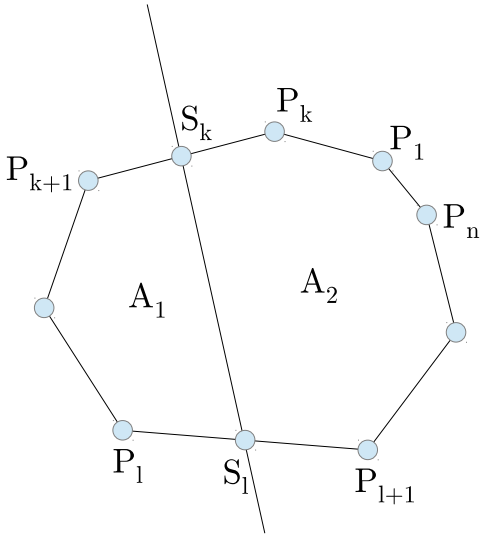
2 P (c) Wie in (b) sei das durch P_1, ..., P_n definierte Polygon konvex. Zusätzlich sind noch die Parameter m und b einer Geraden $y = mx + b$ gegeben, die das Polygon in zwei Teile A_1 und A_2 teilt.



Beschreiben Sie einen effizienten Algorithmus, der die Resultate aus dem Aufgabenteil (b) benutzt, um die Flächeninhalte der Teile A_1 und A_2 zu berechnen. Geben Sie auch die Laufzeit Ihres Verfahrens an.

Lösung: Ist ein Punkt P durch seine Koordinaten x_P und y_P gegeben, dann liegt P unterhalb der Geraden, falls $mx_P + b > y_P$, und oberhalb der Geraden, falls $mx_P + b < y_P$. Im Folgenden nehmen wir der Einfachheit halber o.B.d.A. an, dass sowohl P_1 als auch P_n oberhalb der Geraden liegen. Da nach Voraussetzung die Gerade das Polygon in zwei Teile teilt, muss es einen Punkt P_i geben, der unterhalb der Geraden liegt. Wir iterieren nun über die Punkte $P_1, ..., P_n$ (in dieser Reihenfolge), um den Index k zu finden, für den P_k oberhalb und P_{k+1} unterhalb der Geraden liegt. Für diesen Index berechnen wir den Schnittpunkt S_k, den die Gerade mit

dem Liniensegment $\overline{P_k P_{k+1}}$ bildet. Analog suchen wir den Index l, für den P_l unterhalb und P_{l+1} oberhalb der Geraden liegt und berechnen den entsprechenden Schnittpunkt S_l.



Gesucht werden die Flächeninhalte der durch

$$P_1, ..., P_k, S_k, S_l, P_{l+1}, ..., P_n, P_1$$

und

$$S_k, P_{k+1}, ..., P_l, S_l, S_k$$

definierten Polygone. Diese können mithilfe des Verfahrens aus Aufgabenteil (b) berechnet werden. Da die Schnittpunkte durch einfaches Durchlaufen der Punkte $P_1, ..., P_n$ ermittelt werden können und die Zeit zur Berechnung der Flächeninhalte linear in n ist, beträgt die Gesamtlaufzeit noch immer $\Theta(n)$.