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
Yann Disser

Musterlösung zur Prüfung
Datenstrukturen und Algorithmen
D-INFK

9. August 2012

Aufgabe 1

1 P a) Kreuzen Sie bei jeder der folgenden Datenstrukturen an, ob der in ihr verwendete Baum immer logarithmische Höhe hat (balanciert) oder nicht (unbalanciert):

Natürlicher Suchbaum	☐ BALANCIERT	/ ☒ UNBALANCIERT
AVL-Baum	☒ BALANCIERT	/ ☐ UNBALANCIERT
Optimaler Suchbaum	☐ BALANCIERT	/ ☒ UNBALANCIERT
Heap	☒ BALANCIERT	/ ☐ UNBALANCIERT
Intervallbaum	☒ BALANCIERT	/ ☐ UNBALANCIERT
Segmentbaum	☒ BALANCIERT	/ ☐ UNBALANCIERT
Range Tree	☒ BALANCIERT	/ ☐ UNBALANCIERT

1 P b) Führen Sie auf dem gegebenen Array einen Aufteilungsschritt (in-situ, d.h. ohne Hilfsarray) des Sortieralgorithmus Quicksort durch. Benutzen Sie als Pivot das am linken Ende stehende Element im Array.

Wenn das Pivot bis zum Ende am linken Rand belassen und dann in die Mitte getauscht wird:

32	13	19	8	69	34	77	52	17	49	37	26	5

1	2	3	4	5	6	7	8	9	10	11	12	13
17	13	19	8	5	26	32	52	77	49	37	34	69

Wenn das Pivot rausgenommen und erst am Ende wieder eingefügt wird:

1	2	3	4	5	6	7	8	9	10	11	12	13
13	19	8	5	26	17	32	52	77	49	37	34	69

Wenn immer jeweils mit dem Pivot vertauscht wird:

1	2	3	4	5	6	7	8	9	10	11	12	13
5	13	19	8	26	17	32	52	77	49	37	34	69

1 P c) Das untenstehende Array enthält die Elemente eines Min-Heaps in der üblichen Form gespeichert. Wie sieht das Array aus, nachdem das Minimum entfernt wurde und die Heap-Bedingung wieder hergestellt wurde?

5	7	21	13	45	36	67	22	17	49	87	86	42

1	2	3	4	5	6	7	8	9	10	11	12	13
7	13	21	17	45	36	67	22	42	49	87	86	

1 P d) Fügen Sie die Schlüssel 26,37,33,45,49,29,10,21,7 in dieser Reihenfolge mittels Offenem Hashing in die folgende Hashtabelle ein, und benutzen Sie dabei Double Hashing. Die zu verwendende Hash-Funktion ist $h(k) = k \mod 13$, und für das Sondieren soll die Hashfunktion $h\prime(k) = 1 + (k \mod 11)$ benutzt werden.

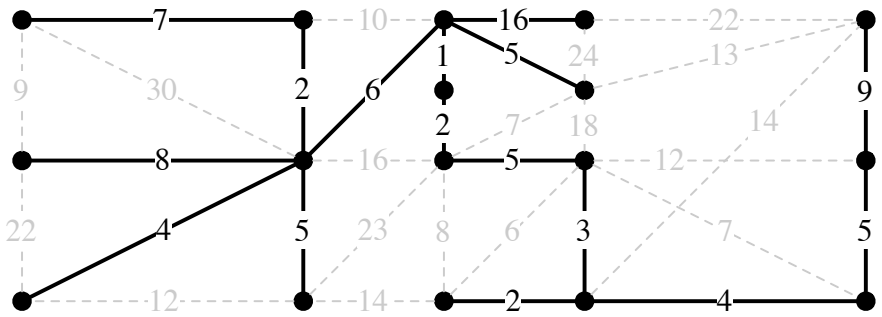
Falls $h(k) - j \cdot h'(k) \mod 13$ **verwendet wird**:
$h(26) = 0$: ok
$h(37) = 11$: ok
$h(33) = 7$: ok
$h(45) = 6$: ok
$h(49) = 10$: ok
$h(29) = 3$: ok
$h(10) = 10$: kollision $\rightarrow h(10) - h'(10) \mod 13 = 12$: ok
$h(21) = 8$: ok
$h(7) = 7$: kollision $\rightarrow h(7) - h'(7) \mod 13 = 12$: kollision $\rightarrow h(7) - 2 \cdot h'(7) \mod 13 = 4$: ok

0	1	2	3	4	5	6	7	8	9	10	11	12
26			29	7		45	33	21		49	37	10

Falls $h(k) + j \cdot h'(k) \mod 13$ **verwendet wird**:
$h(26) = 0$: ok
$h(37) = 11$: ok
$h(33) = 7$: ok
$h(45) = 6$: ok
$h(49) = 10$: ok
$h(29) = 3$: ok
$h(10) = 10$: kollision $\rightarrow h(10) + h'(10) \mod 13 = 8$: ok
$h(21) = 8$: kollision $\rightarrow h(21) + h'(21) \mod 13 = 6$: kollision $\rightarrow h(21) + 2 \cdot h'(21) \mod 13 = 4$: ok
$h(7) = 7$: kollision $\rightarrow h(7) + h'(7) \mod 13 = 2$: ok

0	1	2	3	4	5	6	7	8	9	10	11	12
26		7	29	21		45	33	10		49	37	

1 P e) Markieren Sie die Kanten eines minimalen Spannbaums in folgendem Graphen:

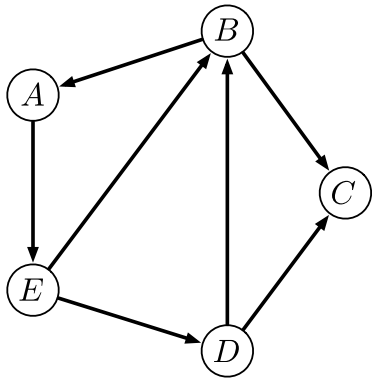


1 P f) Wieviele Kanten kann ein gerichteter Graph mit n Knoten maximal haben, wenn er eine topologische Sortierung besitzen soll?

Wir schauen uns die topologische Sortierung eines Graphen an. Rückwärtskanten darf es nicht geben, also interessiert uns die Zahl der Vorwärtskanten. Maximimal kann jeder Knoten eine Kante zu jedem Nachfolger in der Sortierung haben, also ist die maximale Zahl Kanten

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

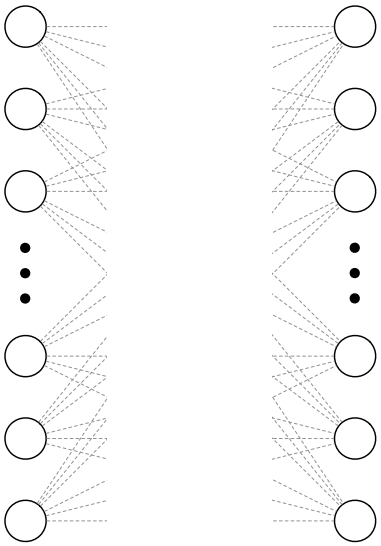
1 P g) Geben Sie jeweils eine Reihenfolge an, in der die Knoten des folgenden Graphen von einer Breitensuche bzw. Tiefensuche mit Startknoten A besucht werden können.



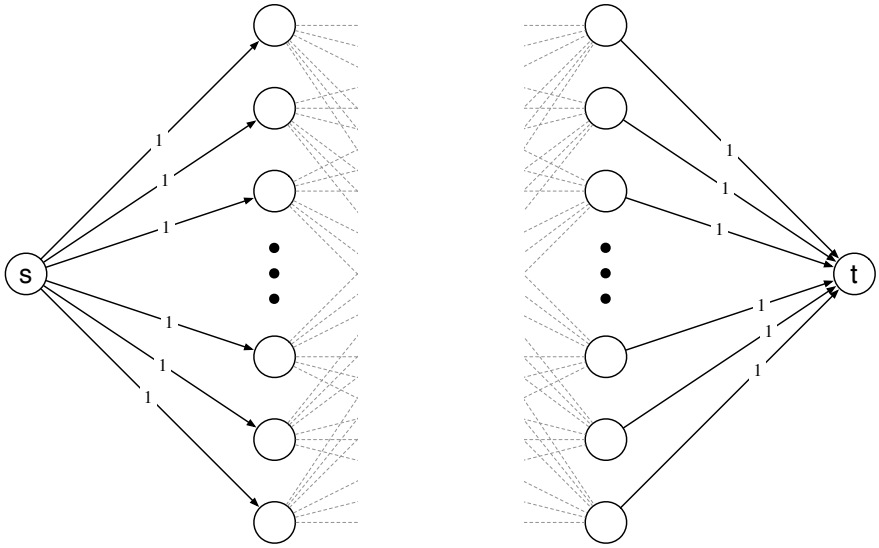
Alle möglichen Breitenordnungen: $AEBDC$; $AEDBC$
Alle möglichen Tiefenordnungen: $AEBCD$; $AEDCB$; $AEDBC$

1 P

h) Wir betrachten einen beliebigen bipartiten Graphen G (die Knoten links bilden die eine Knotenmenge, die Knoten rechts bilden die andere):

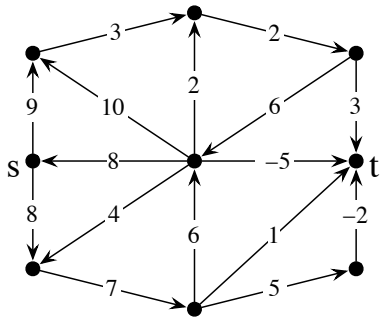


Fügen Sie eine Quelle, eine Senke und zusätzliche Kanten mit Kapazitäten so zu G hinzu, dass ein ganzzahliger, grösstmöglicher Fluss im resultierenden Graphen genau die Kanten eines maximalen Matchings in G verwendet. Für die Flussberechnung dürfen Sie annehmen, dass alle Kanten in G Kapazität 1 haben und von links nach rechts gerichtet sind.



1 P

i) Nennen Sie den Namen eines möglichst effizienten Algorithmus, der geeignet ist einen kürzesten Pfad von s zu t im unten gezeichneten Graphen zu bestimmen. Welche Laufzeit hat dieser Algorithmus im Allgemeinen für Graphen mit n Knoten und m Kanten?



Wegen der negativen Kantengewichte kann Dijkstra's Algorithmus nicht verwendet werden. Stattdessen sollte der Algorithmus von Bellman-Ford verwendet werden. Seine Laufzeit ist $\mathcal{O}(mn)$.

Aufgabe 2

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

Lösung:

Es gilt $\log(n^2) = \Theta(\log n)$, $1000n = \Theta(n)$, $n^2 + 1000 = \Theta(n^2)$. Außerdem ist $5^{\sqrt{n}} < n! < n^n$ für $n \geq 5$ und daher $5^{\sqrt{n}} \in \mathcal{O}(n!)$ und $n! \in \mathcal{O}(n^n)$.

Die einzige gültige Reihenfolge ist:

$$\log(n^2), (\log n)^2, \sqrt{n}, 1000n, n^2 + 1000, n^{1000}, 5^{\sqrt{n}}, n!, n^n$$

3 P b) Gegeben ist die folgende Rekursionsgleichung:

$$T(n) := \begin{cases} 2 + 3T(\frac{n}{7}) & n > 1 \\ 2 & n = 1 \end{cases}$$

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 7 ist, definieren wir zunächst $k \in \mathbb{N}$ so dass $n = 7^k$, also $k = \log_7 n$. Wir teleskopieren um auf eine Formel für $T(n)$ zu kommen:

$$\begin{aligned} T(n) &= 2 + 3T(n/7) \\ &= 2 + 3 \cdot 2 + 3^2 T(n/7^2) \\ &= 2 + 3 \cdot 2 + 3^2 \cdot 2 + 3^3 T(n/7^3) \\ &= \dots \\ &= 2 \sum_{i=0}^{k-1} 3^i + 2 \cdot 3^k \\ &= 2 \cdot \frac{3^k - 1}{3 - 1} + 2 \cdot 3^k \\ &= 3^k - 1 + 2 \cdot 3^k \\ &= 3^{k+1} - 1. \end{aligned}$$

Wir beweisen nun unsere Annahme durch vollständige Induktion über k.

Induktionsanfang ($k = 0$): Es gilt $T(7^k) = T(1) = 2 = 3^{0+1} - 1$.
Induktionsannahme: Für ein $k \in \mathbb{N}_0$ sei $T(7^k) = 3^{k+1} - 1$.
Induktionsschritt ($k \to k + 1$):

$$\begin{aligned} T(7^{k+1}) &= 2 + 3T(7^k) \\ &\overset{\text{Ind. Ann.}}{=} 2 + 3(3^{k+1} - 1) \\ &= 3^{k+2} - 1. \end{aligned}$$

1 P

a) Geben Sie die asymptotische Laufzeit des folgenden Codefragmentes abhängig von $n \in \mathbb{N}$ (möglichst knapp) in Θ-Notation an. Sie müssen Ihre Antwort nicht begründen.

```
for(int i = 0; i < n; ++i)
  for(int j = 0; j < i/2; ++j)
    ;
```

Lösung:

Wir unterschätzen zunächst die Laufzeit: In den letzten $n/2$ Durchläufen der äusseren Schleife läuft die innere jeweils mindestens $n/4$ mal durch. Wir haben also mindestens $\frac{n}{2} \cdot \frac{n}{4} = \Omega(n^2)$ viele Durchläufe.
Nun überschätzen wir: Die innere Schleife läuft immer weniger als n mal durch. Wir haben also weniger als $n \cdot n = \mathcal{O}(n^2)$ viele Durchläufe.
Insgesamt haben wir also eine Laufzeit von $\Theta(n^2)$.

1 P

b) Geben Sie die asymptotische Laufzeit des folgenden Codefragmentes abhängig von $n \in \mathbb{N}$ (möglichst knapp) in Θ-Notation an. Sie müssen Ihre Antwort nicht begründen.

```
for(int i = 0; i < n*n; ++i)
  for(int j = 1; j <= i; j *= 3)
    ;
```

Lösung:

Wir unterschätzen zunächst die Laufzeit: In den letzten $n^2/2$ Durchläufen der äusseren Schleife läuft die innere jeweils mindestens $\log_3 \frac{n^2}{2}$ mal durch. Wir haben also mindestens $\frac{n^2}{2} \cdot \log_3 \frac{n^2}{2} = \Omega(n^2 \log n)$ viele Durchläufe.
Nun überschätzen wir: Die innere Schleife läuft immer weniger als $\log n^2$ mal durch. Wir haben also weniger als $n^2 \log n^2 = \mathcal{O}(n^2 \log n)$ viele Durchläufe.
Insgesamt haben wir also eine Laufzeit von $\Theta(n^2 \log n)$.

1 P

c) Geben Sie die asymptotische Laufzeit der folgenden Funktion abhängig von $n \in \mathbb{N}$ (möglichst knapp) in Θ-Notation an. Sie müssen Ihre Antwort nicht begründen.

```
int f(int n)
{
  if(n <= 0)
    return 1;
  else
    return f(n-1) + f(n-1);
}
```

Lösung:

Die Anzahl Aufrufe ist gegeben durch

$$T(n) = \begin{cases} 2 \cdot T(n-1), & \text{falls } n > 0 \\ 1, & \text{sonst.} \end{cases}$$

Offensichtlich gilt $T(n) = \Theta(2^n)$.

Aufgabe 3

Wir betrachten ein Spiel, bei dem Münzen in einer Reihe von links nach rechts ausliegen. Jede Münze hat einen ganzzahligen Wert und die anfängliche Anzahl Münzen n ist durch 3 teilbar. In jedem Zug darf der Spieler entweder die Münze am linken oder die am rechten Ende der Reihe nehmen (und behalten), aber muss dafür zwei Münzen von dem anderen Ende weglegen. Die weggelegten Münzen sind für den Spieler verloren. Nimmt der Spieler beispielsweise die Münze am linken Ende, muss er die zwei Münzen am rechten Ende weglegen. Das Ziel des Spielers ist es, den Wert der genommenen Münzen zu maximieren.

Beispiel: Im folgenden Beispiel mit $n = 21$ kann der Spieler einen Wert von maximal 803 nehmen. Dazu muss er zuerst einmal vom rechten Ende nehmen, dann einmal vom linken, dann zweimal vom rechten und schliesslich dreimal vom linken. Jede andere Strategie hat (natürlich) die gleiche Anzahl Züge ($n/3 = 7$), erreicht aber einen kleineren Wert.

20	20	500	20	20	20	20	100	100	100	5	5	5	5	5	5	1	1	1	1	1

5 P

a) Entwerfen Sie einen möglichst effizienten Algorithmus nach dem Prinzip der dynamischen Programmierung, der für eine gegebene Münzreihe den maximal für den Spieler erreichbaren Wert berechnet. Beschreiben Sie Ihr dynamisches Programm und geben sie die Laufzeit des Algorithmus an.

Lösung:

Definition der DP Tabelle: Wir verwenden eine Tabelle T der Grösse $n \times (n/3 + 1)$. Der Eintrag an der Stelle $T[i][l]$ enthält den maximal erreichbaren Wert falls die Reihe bei Münze i anfangen würde und Länge $3l$ hätte.

Berechnung eines Eintrags: Wir initialisieren $T[i][0] \leftarrow 0$ für alle i, denn wenn die Zahlenreihe Länge 0 hat, kann nur der Wert 0 erreicht werden. Der Eintrag $T[i][l]$, mit $l > 0$, berechnet sich aus den beiden Einträgen $T[i+1][l-1]$ und $T[i+2][l-1]$. Denn es gibt zwei Möglichkeiten mit einer Zahlenreihe der Länge $3l$ umzugehen: Entweder wir nehmen die erste Münze und verbleiben mit einer Reihe der Länge $3(l-1)$ beginnend mit der zweiten Münze, oder wir nehmen die letzte Münze und verbleiben mit einer Reihe der Länge $3(l-1)$ beginnend mit der dritten Münze. Wenn die Münzwerte in einem Array 'coins' gespeichert sind, haben wir also

$$T[i][l] \leftarrow \begin{cases} 0, & \text{falls } l = 0 \\ \max(\text{coins}[i] + T[i+1][l-1], \text{coins}[i+3l-1] + T[i+2][l-1]), & \text{sonst.} \end{cases}$$

Berechnungsreihenfolge: Da jeder Tabelleneintrag nur von Einträgen mit kleinerem l abhängt, können wir die Einträge einfach aufsteigend nach Länge berechnen.

Auslesen der Lösung: Die Lösung steht am Ende in Eintrag $T[0][n/3]$.

Laufzeit: Die Laufzeit ist $\Theta(n^2)$, da jeder Eintrag mit konstantem Aufwand bestimmt wird.

Beispielcode:

```
int maxValue(const vector<int>& coins)
{
  int n = coins.size();
  vector< vector<int> > T(n, vector<int>(n/3 + 1, 0));

  for(int l = 1; l <= n/3; ++l)
    for(int i = 0; i <= n - 3*l; ++i)
      T[i][l] = max(coins[i] + T[i+1][l-1], coins[i+3*l-1] + T[i+2][l-1]);
```

```
  return T[0][n];
}
```

1 P b) Beschreiben Sie kurz, wie Sie Ihren Algorithmus anpassen können, wenn er nicht nur den maximalen Wert, sondern auch die Zugfolge einer bestmöglichen Strategie bestimmen soll. (Im obigen Beispiel also so etwas wie ,,RLRRLLL“.)

Lösung:

Wir können die Zugabfolge einfach bestimmen, indem wir die Züge vom Eintrag $T[0][n/3]$ ausgehend zurückverfolgen. Um zu sehen, welchen Zug wir einer Situation gemacht haben, schauen wir an die entsprechende Position der Tabelle $T[i][l]$. Falls $T[i][l] = \text{coins}[i] + T[i+1][l-1]$, haben wir die erste Münze genommen und geben entsprechend ,,L“ aus. Ansonsten haben wir die letzte Münze genommen und geben entsprechend ,,R“ aus.

3 P c) Wir betrachten eine erweiterte Version des Spiels mit einem Spieler A und einem Gegenspieler B. Die Spieler sind abwechselnd am Zug, wobei A das Spiel beginnt. Immer wenn Spieler A am Zug ist, nimmt er *eine* Münze vom linken oder vom rechten Ende. Spieler B nimmt in jedem Zug *zwei* Münzen vom linken oder *zwei* Münzen vom rechten Ende.

Wir interessieren uns für eine gute Strategie für Spieler A, unabhängig von B's Strategie. Wir wollen also bestimmen, welchen maximalen Wert Spieler A **garantiert** erreichen kann, unabhängig von B's Spielweise.

Beschreiben Sie einen möglichst effizienten Algorithmus, der für eine gegebene Münzreihe den maximalen, garantiert erreichbaren Wert für Spieler A berechnet. Geben sie die Laufzeit Ihres Algorithmus an.

Lösung:

Wir verwenden erneut dynamische Programmierung, diesmal allerdings mit zwei Tabellen (es würde auch mit einer gehen, aber mit zweien ist es übersichtlicher). Wir gehen wieder davon aus, dass die Münzwerte in einem Array 'coins' abgelegt sind.

Definition der DP Tabellen: Wir verwenden zwei Tabellen T_A und T_B der Grösse $(n+1)\times(n+1)$. Der Eintrag an der Stelle $T_X[i][l]$, $X \in \{A, B\}$, enthält den maximal für A erreichbaren Wert falls die Reihe bei Münze i anfangen würde und Länge l hätte, wenn Spieler X am Zug ist. Für Spieler A interessieren uns eigentlich nur Einträge mit $l = 0 \mod 3$ und für Spieler B nur Einträge mit $l = 2 \mod 3$, der Einfachheit halber berechnen wir dennoch alle Einträge. (Im Grunde würde eine einzelne Tabelle der Grösse $n \times (2n/3)$ ausreichen.)

Berechnung eines Eintrags: Wir initialisieren $T_A[i][l] \leftarrow T_B[i][l] \leftarrow 0$ für alle i und $l < 2$, denn wenn die Zahlenreihe Länge 0 hat, kann nur der Wert 0 erreicht werden, der Fall mit Länge 1 tritt später nie auf und ist daher irrelevant.

Der Eintrag $T_A[i][l]$, mit $l > 1$, berechnet sich aus den beiden Einträgen $T_B[i+1][l-1]$ und $T_B[i][l-1]$. Denn es gibt zwei Möglichkeiten für Spieler A mit einer Zahlenreihe der Länge l umzugehen: Entweder er nimmt die erste Münze und Spieler B macht mit einer Reihe der Länge $l-1$ beginnend mit der zweiten Münze weiter, oder er nimmt die letzte Münze und Spieler B macht mit einer Reihe der Länge $l-1$ beginnend mit der ersten Münze weiter. Von diesen beiden Möglichkeiten wählt Spieler A natürlich die **bessere**

$$T_A[i][l] \leftarrow \begin{cases} 0, & \text{falls } l < 2 \\ \max(\text{coins}[i] + T_B[i+1][l-1], \text{coins}[i+l-1] + T_B[i][l-1]), & \text{sonst.} \end{cases}$$

Der Eintrag $T_B[i][l]$, mit $l > 1$, berechnet sich aus den beiden Einträgen $T_A[i+1][l-2]$ und $T_A[i][l-2]$. Denn es gibt zwei Möglichkeiten für Spieler B mit einer Zahlenreihe der Länge

l umzugehen: Entweder er nimmt die ersten beiden Münze und Spieler A macht mit einer Reihe der Länge $l-2$ beginnend mit der dritten Münze weiter, oder er nimmt die letzten beiden Münze und Spieler A macht mit einer Reihe der Länge $l-2$ beginnend mit der ersten Münze weiter. Von diesen beiden Möglichkeiten wählt Spieler B die **schlechtere**, also die mit niedrigerem Wert. Das macht Sinn, da Spieler B genau die Münzen bekommt, die Spieler A *nicht* bekommt. Statt also seinen eigenen Wert zu maximieren, kann Spieler B genausogut A's Wert minimieren:

$$T_B[i][l] \leftarrow \begin{cases} 0, & \text{falls } l < 2 \\ \min(T_A[i+2][l-2], T_A[i][l-2]), & \text{sonst.} \end{cases}$$

Berechnungsreihenfolge: Da jeder Tabelleneintrag nur von Einträgen mit kleinerem l abhängt, können wir die Einträge einfach aufsteigend nach Länge berechnen.

Auslesen der Lösung: Die Lösung steht am Ende in Eintrag $T_A[0][n]$.

Beispielcode:

```
int maxValueA(const vector<int>& coins)
{
  int n = coins.size();
  vector< vector<int> > TA(n+1, vector<int>(n+1,0));
  vector< vector<int> > TB(n+1, vector<int>(n+1,0));

  for(int l = 2; l <= n; ++l)
    for(int i = 0; i <= n-l; ++i)
    {
      TA[i][l] = max(coins[i] + TB[i+1][l-1], coins[i+l-1] + TB[i][l-1]);
      TB[i][l] = min(TA[i+2][l-2], TA[i][l-2]);
    }

  return TA[0][n];
}
```

Aufgabe 4

Sie sollen einen Algorithmus zur Optimierung von mehrtägigen Fahrradtouren entwerfen, bei denen über Nacht gezeltet wird. Eine Tour besteht aus n Streckenabschnitten und $n+1$ Orten, wobei der i-te Streckenabschnitt die Orte i und $i+1$ verbindet. Gegeben ist eine solche Tour, ein Zeitraum T (in Tagen), den die Tour dauern soll, und eine Wettervorhersage für alle Bereiche der Tour. Die Wettervorhersage beschreibt die Niederschlagsmenge (Regen) auf jedem Streckenabschnitt für jeden Tag und die Niederschlagsmenge für jede Nacht an jedem Ort. Eine Instanz mit $n = 3$ und $T = 6$ sieht also beispielsweise so aus:

	Abschnitt 1	Abschnitt 2	Abschnitt 3
Tag 1	2mm	3mm	1mm
Tag 2	1mm	4mm	0mm
Tag 3	3mm	0mm	8mm
Tag 4	3mm	7mm	5mm
Tag 5	10mm	1mm	2mm
Tag 6	5mm	5mm	5mm

	Ort 1	Ort 2	Ort 3	Ort 4
Nacht $1 \to 2$	0mm	2mm	0mm	0mm
Nacht $2 \to 3$	1mm	0mm	2mm	1mm
Nacht $3 \to 4$	0mm	1mm	15mm	0mm
Nacht $4 \to 5$	2mm	0mm	15mm	4mm
Nacht $5 \to 6$	0mm	4mm	0mm	3mm

Gesucht ist ein Plan, an welchen Tagen die Streckenabschnitte gefahren werden sollten, um die **Summe** der Niederschläge (gemäss Vorhersage) möglichst klein zu halten. Die Streckenabschnitte sollen in der gegebenen Reihenfolge befahren werden (also Abschnitt i vor Abschnitt $i+1$), aber es können *beliebig viele Abschnitte pro Tag* eingeplant werden. Ausserdem muss *jede* der Nächte an einem Ort verbracht werden, es können aber mehrere Nächte am gleichen Ort verbracht werden. Wenn der erste Abschnitt also für Tag j geplant ist, muss vorher erst $j-1$ mal an Ort 1 übernachtet werden. Genauso muss am Ende $T - j$ mal an Ort $n + 1$ übernachtet werden, wenn der letzte Abschnitt für Tag j geplant ist. Der Gesamtniederschlag für eine geplante Fahrradtour ist die Summe der Niederschläge für jeden Streckenabschnitt und für jede Übernachtung an einem Ort.

Beispiel: Im obigen Beispiel kommt die beste Lösung mit 8mm Niederschlag aus. Dazu wird zunächst an Ort 1 übernachtet (0mm), an Tag 2 wird Streckenabschnitt 1 gefahren (1mm), dann wird dreimal an Ort 2 übernachtet (0+1+0=1mm), an Tag 5 werden die Abschnitte 2 und 3 gefahren (1+2=3mm), und schliesslich wird noch einmal an Ort 4 übernachtet (3mm). Wenn wir stattdessen zum Beispiel alle Abschnitte gleich an Tag 1 fahren (2+3+1=6mm), müssen wir fünf mal an Ort 4 übernachten (0+1+0+4+3=8mm), und haben also insgesamt einen Niederschlag von 14mm.

4 P a) Konstruieren Sie einen gerichteten Graphen G, so dass ein kürzester Weg in G einer Fahrradtour mit minimalem Niederschlag entspricht. Führen Sie dazu für jeden Zustand des Systems einen Knoten in G ein und definieren Sie (gerichtete und gewichtete) Kanten, die den Zustandsübergängen entsprechen. Beschreiben Sie Ihre Konstruktion und leiten Sie daraus einen möglichst effizienten Algorithmus zur Planung von Fahrradtouren mit minimalem Niederschlag ab.

Lösung:

Ein Zustand ist ein Tupel bestehend aus der aktuellen Tag und dem aktuellen Ort. Für jeden solchen Zustand führen wir einen Knoten ein. Jeder Knoten (t, o) hat maximal zwei Kanten: Eine Kante zum nächsten Tag (falls dieser existiert) am selben Ort, also zum Knoten $(t+1, o)$; die Kante hat ein Gewicht, dass der Niederschlagsmenge in Nacht $t \to t+1$ am Ort o entspricht. Und eine Kante zum nächsten Ort (falls dieser existiert) am selben Tag, also zum Knoten $(t, o+1)$; die Kante hat ein Gewicht, dass der Niederschlagsmenge an Tag t auf Streckenabschnitt o entspricht.

Eine Fahrradtour mit minimalem Niederschlag entspricht einem kürzesten Weg von $(1, 1)$ zu $(T, n + 1)$. Wir können Dijkstra's Algorithmus benutzen um einen solchen Weg zu ermitteln, oder ein einfaches dynamisches Programm.

2 P b) Welche Laufzeit hat Ihr Algorithmus in Abhängigkeit von n und T?

Lösung:

Die Anzahl Knoten ist circa nT (genauer: $(n + 1)T$) und die Anzahl Kanten ist circa $2nT$ (genauer: $2n(T-1)+n+T-1 = 2nT+T-n-1$). Die Laufzeit von Dijkstra's Algorithmus ist daher $\mathcal{O}(nT \log(nT))$. Mit einem dynamischen Programm lässt sich sogar $\mathcal{O}(nT)$ erreichen.

3 P c) Nehmen Sie nun an, dass nicht mehr als 3 Streckenabschnitte pro Tag gefahren werden können. Wie können Sie die Konstruktion von G anpassen, so dass ein Plan mit minimalem Niederschlag und maximal 3 Streckenabschnitten pro Tag einem kürzesten Weg in G entspricht (und umgekehrt)? Wie verändert sich dadurch die Laufzeit Ihres Algorithmus?

Lösung:

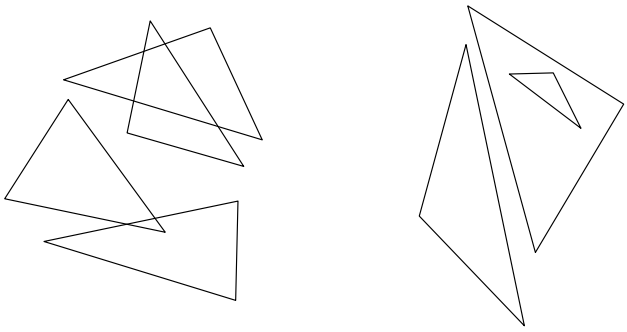
Es gibt zwei einfache Möglichkeiten die Anzahl Streckenabschnitte pro Tag zu begrenzen.

Möglichkeit 1: Wir verwenden im Wesentlichen die gleichen Kanten, erweitern aber Zustände auf Tripel (t, o, k), wobei $0 \leq k \leq 3$ die Anzahl an Tag t bereits gefahrener Streckenabschnitte entspricht. Wir haben nun Kanten von (t, o, k) zu $(t + 1, o, 0)$, falls $t < T$, und von (t, o, k) zu $(t, o + 1, k + 1)$, falls $o < n + 1$ und $k < 3$. Die Kantengewichte bleiben analog. Allerdings müssen wir einen separaten Zielknoten einführen, der von den Knoten $(n+1, T, 0)$, $(n+1, T, 1)$, $(n+1, T, 2)$, $(n+1, T, 3)$ mitels einer Kante mit Kosten 0 erreichbar ist. Wir haben (ungefähr) viermal so viele Knoten wie bislang, also nur einen konstanten Faktor mehr. Die asymptotische Laufzeit bleibt entsprechend die gleiche.

Möglichkeit 2: Wir verwenden die gleichen Zustände, ändern aber die Kanten. Für Zustand (t, o) verwenden wir nun Kanten nach $(t + 1, o)$, $(t + 1, o + 1)$, $(t + 1, o + 2)$, $(t + 1, o + 3)$, natürlich nur wenn der entsprechende Tag und die entsprechenden Orte jeweils existieren. Die Kanten zu $(t + 1, o)$ hat das gleiche Gewicht wie zuvor. Die Kante zu $(t + 1, o + 1)$ verwendet den Niederschlag von Abschnitt o an Tag t summiert mit dem Niederschlag in Nacht $t \to t+1$ an Ort $o + 1$. Die Kante zu $(t + 1, o + 2)$ verwendet die Niederschläge von Abschnitt o und $o + 1$ an Tag t summiert mit dem Niederschlag in Nacht $t \to t + 1$ an Ort $o + 2$. Die Kante zu $(t + 1, o + 3)$ verwendet die Niederschläge von Abschnitt o, $o + 1$, und $o + 2$ an Tag t summiert mit dem Niederschlag in Nacht $t \to t + 1$ an Ort $o + 3$. Wir haben zweimal so viele Kanten wie bislang, also nur einen konstanten Faktor mehr. Die asymptotische Laufzeit bleibt entsprechend die gleiche.

Aufgabe 5

Gegeben ist eine Menge von Dreiecken in der Ebene. Jedes Dreieck ist durch die Koordinaten seiner Eckpunkte gegeben, wobei wir der Einfachheit halber annehmen, dass keine zwei Eckpunkte die gleiche x-Koordinate haben. Wir möchten herausfinden, ob die Menge zwei Dreiecke enthält, die sich überlappen.



3 P a) Wir ermitteln zunächst, ob es Dreiecke gibt, deren Ränder sich schneiden (linkes Bild). Beschreiben Sie einen möglichst effizienten *Scanlinealgorithmus*, der dies bestimmt. Sie dürfen davon ausgehen, dass sich die Ränder keiner zwei Dreiecke berühren ohne sich zu schneiden. Welche Laufzeit hat Ihr Verfahren?

Lösung:

Wir können hier einfach das Standardverfahren für Segmentschnitte verwenden. Dazu führen wir die Kanten jedes Dreiecks als Segmente ein. Etwas aufpassen muss man nur, dass Schnitte der Segmente des gleichen Dreiecks ignoriert werden. Ansonsten können wir abbrechen, sobald ein Schnitt gefunden wird. Die Laufzeit ist $\Theta(n \log n)$, wobei n die Zahl der Dreiecke bezeichnet.

6 P b) Falls sich die Ränder keiner zwei Dreiecke schneiden, müssen wir noch feststellen, ob es ein Dreieck gibt, das in einem anderen enthalten ist (rechtes Bild). Beschreiben Sie einen möglichst effizienten *Scanlinealgorithmus*, der bestimmt, ob dies der Fall ist. Sie dürfen davon ausgehen, dass sich die Ränder keiner zwei Dreiecke *berühren oder schneiden*. Welche Laufzeit hat Ihr Verfahren?

Lösung:

Haltepunkte: Wir verwenden ein vertikale Scanline, die von links nach rechts läuft. Für jedes Dreieck verwenden wir die Ecken mit kleinster und grösster x-Koordinate jeweils als Haltepunkt.

Scanline-Datenstruktur: Da sich keine zwei Dreiecke kreuzen, denn dann können die Dreiecke eindeutig nach y-Koordinate verglichen werden. Zwar haben die Dreiecke y-Koordinaten die in x-Richtung variieren, die relative Ordnung der Dreiecke bleibt jedoch gleich. Daher können wir als Scanline-Datenstruktur einen AVL-Baum verwenden, der in y-Richtung geordnet ist.

Aktualisierung: Wenn wir einen Haltepunkt erreichen, der dem Beginn eines Dreiecks entspricht, suchen wir die geeignete Einfügstelle für das Dreieck im AVL-Baum. Dazu müssen wir das neue Dreieck an jedem Knoten mit dem Dreieck des Knotens vergleichen. Das machen wir einfach, indem wir für den soeben mit der Scanline erreichten Eckpunkt des neuen Dreiecks prüfen, ob dieser im Dreieck des Knotens liegt, oder nicht. Falls ja, ist das neue Dreieck in ihm komplett enthalten, da sie sich ja nicht kreuzen, und wir können abbrechen. Ansonsten schauen wir, ob wir über oder unter dem Dreieck liegen und gehen entsprechend im

AVL-Baum weiter. Wenn wir bei einem Blatt ankommen, fügen wir das Dreieck wie gewohnt in den Baum ein.

Wenn wir einen Haltepunkt erreichen, der dem Ende eines Dreiecks entspricht, löschen wir das entsprechende Dreieck einfach aus dem Baum. Dazu müssen wir es zunächst analog zum Einfügen suchen.

Auslesen der Lösung: Unser Verfahren bricht ab, falls es ein Dreieck gibt, das in einem anderen enthalten ist. Ansonsten terminiert es regulär, nachdem alle Haltepunkte behandelt wurden.

Laufzeit: Wir bezeichnen die Zahl Dreiecke mit n. Das Sortieren der Eckpunkte benötigt eine Laufzeit von $\Theta(n \log n)$. Das Einfügen/Löschen im AVL-Baum jeweils $\Theta(\log n)$ für alle $\Theta(n)$ Haltepunkte, denn wir können einfach (per Kreuzprodukt oder Vergleich der Geradengleichungen) in konstanter Zeit bestimmen, ob ein Punkt über oder unter einer Geraden liegt – und damit auch, ob ein Punkt über, unter, oder in einem Dreieck liegt. Die Laufzeit ist daher insgesamt $\Theta(n \log n)$.