Prof. Dr. Ir. Joost-Pieter Katoen

Christian Dehnert, Friedrich Gretz, Benjamin Kaminski, Thomas Ströder

Tutoraufgabe 1 (Bellman-Ford Algorithmus):

a) Passen Sie die Implementierung des Bellman-Ford-Algorithmus, die Sie in der Vorlesung kennengelernt haben (Vorlesung 17, Folie 13), so an, dass der kürzeste Pfad zwischen zwei Knoten i und j ausgegeben (nicht zurückgegeben!) wird. Sie dürfen davon ausgehen, dass der übergebene Graph keine negativen Zykel besitzt.

Ihre Funktion soll die folgende Signatur haben:

```
void bellFord(List adjLst[n], int n, int i, int j)
```

Zur Ausgabe können Sie annehmen, dass eine Funktion `ausgabe` mit folgender Signatur existiert:

```
void ausgabe(String text)
```

Außerdem können Sie annehmen, dass `int`-Werte automatisch nach `String` konvertiert werden können.

b) Einem Studenten gefällt die Laufzeit des Bellman-Ford-Algorithmus nicht und er schlägt das folgende alternative Verfahren vor, um Zyklen mit negativem Gesamtgewicht zu erkennen:

Wir benutzen Sharirs Algorithmus, um alle starken Zusammenhangskomponenten zu finden. Dies kostet $\mathcal{O}(|V|+|E|)$. Da $|E| \in \mathcal{O}(|V|^2)$ sind die Kosten damit in $\mathcal{O}(|V|+|V|^2) = \mathcal{O}(|V|^2)$. Für jede starke Zusammenhangskomponente berechnen wir dann die Summe der Gewichte ihrer Kanten. Dies kostet insgesamt $\mathcal{O}(|E|)$ und ist also wiederum in $\mathcal{O}(|V|^2)$. Ist eine dieser Summen negativ, geben wir TRUE aus, sonst FALSE.

Wo liegt der Fehler, den der Student begangen hat?

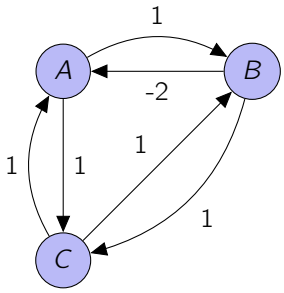
Lösung:

a) Der folgende Algorithmus gibt einen kürzesten Pfad von i nach j aus:

```
void bellFord(List adjLst[n], int n, int i, int j){
    int d[n] = +inf;
    d[i] = 0;

    int parent[n] = -1;

    for (int k = 1; k < n; k++) // n-1 Durchläufe
        for (int v = 0; v < n; v++) // alle Kanten
            foreach (edge in adjLst[v])
                if (d[edge.w] > d[v] + edge.weight) {
                    d[edge.w] = d[v] + edge.weight;
                    parent[edge.w] = v;
                }
    ausgabe("Ein kürzester Weg mit Länge " + d[j] + ":");
    printPfad(parent, j);
}

// printPfad gibt den Pfad rekursiv aus
void printPfad(int parent[n], int node){
    // gibt es weiter Vorgänger, so wird zuerst der Pfad bis zu diesem Knoten ausgegeben
    if(parent[node] != -1){
        printPfad(parent, parent[node]);
        ausgabe(" -> ");
    }
    // Zuletzt wird der Knoten selbst ausgegeben
```

```
        ausgabe(node);
    }
```

b) Eine starke Zusammenhangskomponente ist nicht dasselbe wie ein Zyklus. Betrachten wir den folgenden Graphen G_3:



Die einzige starke Zusammenhangskomponente ist der gesamte (vollständige) Graph. Ihr Gesamtgewicht ist 3. Somit würde das Verfahren des Studenten FALSE ausgeben. Es gibt jedoch einen Zyklus mit negativem Gewicht von A nach B und zurück.

Tutoraufgabe 2 (Dijkstra Algorithmus):

a) Beweisen oder widerlegen Sie die folgenden Aussagen für einen gewichteten, zusammenhängenden, ungerichteten Graphen G:

i) Der vom Dijkstra-Algorithmus berechnete SSSP-Baum ist ein Spannbaum.

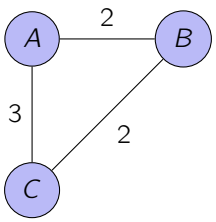
ii) Der vom Dijkstra-Algorithmus berechnete SSSP-Baum ist ein minimaler Spannbaum.

b) Arbeitet der Dijkstra-Algorithmus immer korrekt, wenn man ihn auf einen Graphen mit negativen Gewichten anwendet, der keine Zyklen mit negativem Gesamtgewicht enthält? Begründen Sie Ihre Antwort!

Lösung:

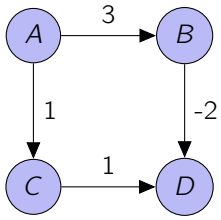
a) i) Da jede hinzugefügte Kante zu einem Knoten führt, der bisher nicht zum Baum gehört hat, kann kein Zyklus entstehen. Da außerdem jede dieser Kanten von einem Knoten ausgeht, der zum bisherigen Baum gehört, ist dieser zusammenhängend. Es handelt sich also tatsächlich um einen Baum. Da der Algorithmus erst terminiert, wenn es keine Randknoten mehr gibt, und der Graph zusammenhängend ist, werden alle Knoten hinzugefügt. Damit ist der SSSP-Baum ein Spannbaum. Die Aussage ist damit gezeigt.

ii) Betrachten wir folgenden Graphen G_4:



Der minimale Spannbaum ist $(\{A, B, C\}, \{\{A, B\}, \{B, C\}\})$ mit dem Gewicht 4. Der SSSP-Baum für den Startknoten A ist hingegen $(\{A, B, C\}, \{\{A, B\}, \{A, C\}\})$ mit dem Gewicht 5. Da $5 > 4$ gilt, ist der SSSP-Baum nicht minimal und die Aussage ist widerlegt.

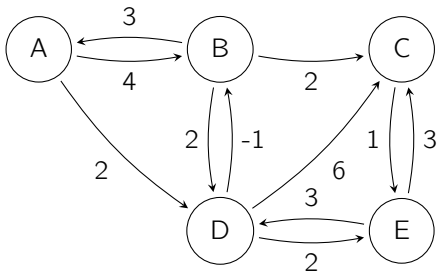
b) Betrachten wir den folgenden Graphen G_5:



Vom Startknoten A aus berechnet der Dijkstra Algorithmus die kürzeste Distanz von A zu D als 2. Dabei ist die tatsächliche kürzeste Distanz 1. Also arbeitet der Algorithmus nicht immer korrekt, wenn er auf Graphen mit negativen Gewichten angewendet wird.

Tutoraufgabe 3 (Bellman-Ford Algorithmus):

Betrachten Sie den folgenden Graphen:



Führen Sie den *Bellman-Ford-Algorithmus* auf diesem Graphen mit dem *Startknoten A* aus. Die Knoten werden dabei in alphabetischer Reihenfolge betrachtet. Sie dürfen den Algorithmus vorzeitig abbrechen, wenn sich keine weiteren Änderungen mehr ergeben. Außerdem dürfen Sie Zellen, deren Werte von Zeile zu Zeile gleich bleiben, leer lassen. Füllen Sie dazu die nachfolgende Tabelle aus:

Geben Sie außerdem an, ob der Algorithmus einen Zyklus mit negativem Gesamtgewicht gefunden hat.

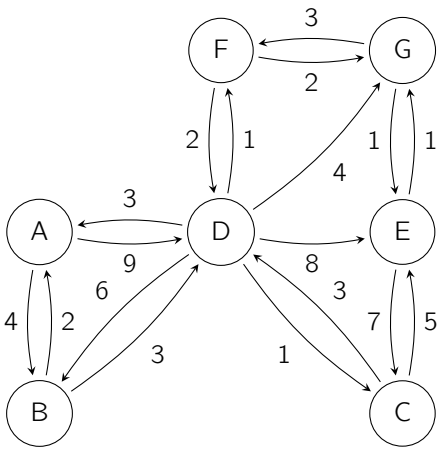
Lösung:

Aktueller Knoten / Entfernung	A	B	C	D	E
–	0	∞	∞	∞	∞
A		4		2	
B			6		
C					7
D		1			4
E					
A					
B			3		

Der Algorithmus hat keinen Zyklus mit negativem Gesamtgewicht gefunden.

Tutoraufgabe 4 (Dijkstra Algorithmus):

Betrachten Sie den folgenden Graphen:



Führen Sie den *Dijkstra* Algorithmus auf diesem Graphen mit dem *Startknoten A* aus. Falls mehrere Knoten für die nächste Iteration zur Wahl stehen, werden die Knoten dabei in alphabetischer Reihenfolge betrachtet. Füllen Sie dazu die nachfolgende Tabelle aus:

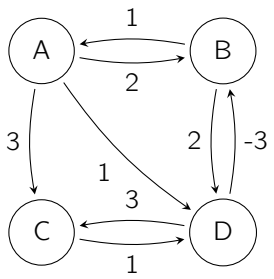
Lösung:

Knoten	A	B	D	C	F	G
B	4	–	–	–	–	–
C	∞	∞	8	–	–	–
D	9	7	–	–	–	–
E	∞	∞	15	13	13	11
F	∞	∞	8	8	–	–
G	∞	∞	11	11	10	–

Die grau unterlegten Zellen markieren, an welcher Stelle für welchen Knoten die minimale Distanz sicher berechnet worden ist.

Aufgabe 5 (Bellman-Ford Algorithmus): (4 Punkte)

Betrachten Sie den folgenden Graphen:



Führen Sie den *Bellman-Ford-Algorithmus* auf diesem Graphen mit dem *Startknoten A* aus. Die Knoten werden dabei in alphabetischer Reihenfolge betrachtet. Sie dürfen den Algorithmus vorzeitig abbrechen, wenn sich keine weiteren Änderungen mehr ergeben. Außerdem dürfen Sie Zellen, deren Werte von Zeile zu Zeile gleich bleiben, leer lassen. Füllen Sie dazu die nachfolgende Tabelle aus:

Geben Sie außerdem an, ob der Algorithmus einen Zyklus mit negativem Gesamtgewicht gefunden hat.

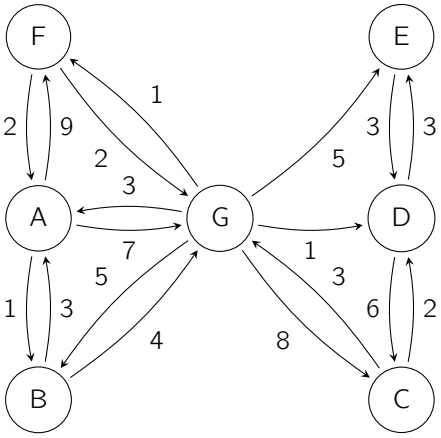
Lösung:

Aktueller Knoten / Entfernung	A	B	C	D
–	0	∞	∞	∞
A		2	3	1
B				
C				
D		-2		
A				
B	-1			0
C				
D		-3		
A			2	
B	-2			-1
C				
D		-4		

Der Algorithmus hat einen Zyklus mit negativem Gesamtgewicht gefunden.

Aufgabe 6 (Dijkstra Algorithmus): (3 Punkte)

Betrachten Sie den folgenden Graphen:



Führen Sie den *Dijkstra* Algorithmus auf diesem Graphen mit dem *Startknoten A* aus. Falls mehrere Knoten für die nächste Iteration zur Wahl stehen, werden die Knoten dabei in alphabetischer Reihenfolge betrachtet. Füllen Sie dazu die nachfolgende Tabelle aus:

Lösung:

Knoten	A	B	G	D	F	E
B	1	–	–	–	–	–
C	∞	∞	13	12	12	12
D	∞	∞	6	–	–	–
E	∞	∞	10	9	9	–
F	9	9	6	6	–	–
G	7	5	–	–	–	–

Die grau unterlegten Zellen markieren, an welcher Stelle für welchen Knoten die minimale Distanz sicher berechnet worden ist.

Aufgabe 7 (Azyklische Graphen): (3 + 5 Punkte)

Sei $G = (V, E, W)$ ein azyklischer, gerichteter, (kanten-)gewichteter Graph.

a) Entwerfen Sie einen Algorithmus, der den längsten Pfad (bezüglich der Gewichte) in G in Zeit $\mathcal{O}(|V|+|E|)$ berechnet.

b) Entwerfen Sie einen Algorithmus, der die Anzahl der Pfade in G in Zeit $\mathcal{O}(|V|+|E|)$ berechnet. Beachten Sie hierbei auch **leere Pfade**.

Lösung:

a) Die einzigen Unterschiede zur Berechnung von kritischen Pfaden (siehe Übung 7) ist, dass die Kanten gewichtet sind und nicht die Knoten und negative Gewichte erlaubt sind.

Der gesuchte Algorithmus könnte wie folgt aussehen. Wir können einen kantengewichteten Graphen in $\mathcal{O}(|V|+|E|)$ in einen knotengewichteten Graphen überführen, indem wir jede Kante durch zwei neue Kanten und einen Zwischenknoten ersetzen, wobei der Zwischenknoten das Gewicht der ursprünglichen Kante erhält. Wir konstruieren also

$$G' = (V' = V \cup \{v_e \mid e \in E\}, E' = E \cup \{(u, v_e), (v_e, v) \mid e = (u, v) \in E\}, W' \colon V' \to \mathbb{R})$$

mit

$$W'(v) = \begin{cases} 0 & v \in V \\ W(e) & v = v_e \end{cases}$$

G' kann in Zeit $\mathcal{O}(|V'| + |E'|) = \mathcal{O}((|V| + |E|) + 2|E|) = \mathcal{O}(|V| + |E|)$ berechnet werden. Anschließend kann in G' der längste Pfad mit folgender Variante (die mit negativen Knotengewichten umgehen kann) des Algorithmus aus der Vorlesung

```
void dfsSearch(List adjL[n], int start,
               int &color[n], int &duration[n],
               int &critDep[n], int &eft[n]) {
  color[start] = GRAY; critDep[start] = -1; int est = -inf;
  if (adjL[start].empty()) {
    // Knoten ohne ausgehende Kante hat est 0
    est = 0;
  } else {
```

```
    foreach (next in adjL[start]) {
      if (color[next] == WHITE) {
        dfsSearch(adjL, n, next, color, paths);
      }
      // da est initial -inf ist, trifft diese Bedingung
      // mindestens für den ersten Nachfolger zu
      if (eft[next] >= est) {
          est = eft[next];
          critDep[start] = next;
      }
    }
  }
  eft = est + duration[start];
  color[start] = BLACK;
}

void critPathExt(List adj[n], int n, int duration[n],
                 int &critDep[n], int &eft[n]) {
  int color[n] = WHITE;
  for (int i = 0; i < n; i++) {
    if (color[i] == WHITE) {
      dfsSearch(adj, i, color, duration, critDep, eft);
    }
  }
}
```

in Laufzeit $\mathcal{O}(|V'| + |E'|) = \mathcal{O}((|V| + |E|) + 2|E|) = \mathcal{O}(|V| + |E|)$ berechnet werden.

b) Wir benutzen folgende Variation des Algorithmus zur Bestimmung kritischer Pfade. Dabei speichert das Array `paths` für jeden Knoten v die Anzahl der Pfade, die v als ersten Knoten haben. Die Funktion `pathCount` startet die Tiefensuchen und addiert die Anzahl der ausgehenden Pfade jedes Knoten v, um die Gesamtzahl aller Pfade in G zu berechnen.

```
void dfsSearch(List adjL[n], int start,
               int &color[n], int &paths[n]) {
  path = 0;
  color[start] = GRAY;
  foreach (next in adjL[start]) {
    if (color[next] == WHITE) {
      dfsSearch(adjL, n, next, color, paths);
    }
    path = path + paths[next];
  }
  // leerer Pfad + Pfade hinter start
  paths[start] = 1 + path;
  color[start] = BLACK;
}

oid pathCount(List adj[n], int n) {
  int color[n] = WHITE;
  int paths[n] = 0;
  int pathCount = 0;
  for (int i = 0; i < n; i++) {
    if (color[i] == WHITE) {
      dfsSearch(adj, i, color, paths);
```

```
        }
        pathCount = pathCount + paths[i];
      }
    }
```

Aufgabe 8 (Matrjoschka-Kartons): (3 + 3 + 5 Punkte)

Betrachten wir einen k-dimensionalen Karton A mit den Abmessungen $(a_1, \ldots, a_k)$. Wir wollen nun einen Matrjoschka-Karton (ineinander verschachtelte Kartons) aus k-dimensionalen Kartons bauen (siehe `http://de.wikipedia.org/wiki/Matrjoschka`). Ein Karton A passt in einen anderen k-dimensionalen Karton B mit Abmessungen $(b_1, \ldots, b_k)$, wenn eine Permutation π von $\{1, \ldots, k\}$ existiert, sodass

$$a_{\pi(1)} < b_1, a_{\pi(2)} < b_2, \ldots, a_{\pi(k)} < b_k.$$

Passt A in B, so schreiben wir $A \prec B$. (Im Fall von 3 Dimensionen stimmt diese Definition mit der "intuitiven" überein, wobei die Permutation die Möglichkeit des Drehens der Kiste widerspiegelt.)

a) Zeigen Sie, dass wenn $A \prec B$ und $B \prec C$, dann auch $A \prec C$.

b) Entwerfen Sie einen Algorithmus, der in $\mathcal{O}(k^2)$ prüft, ob $A \prec B$.

c) Es seien nun n k-dimensionale Kartons $\{A_1, \ldots, A_n\}$ gegeben. Da wir den größtmöglichen Matrjoschka-Karton (hinsichtlich der Anzahl der Kartons) bauen wollen, sind wir an der maximalen Schachtelungstiefe interessiert. Geben Sie dazu einen Algorithmus an, der die Länge der längsten Folge $\langle A_{i_1}, A_{i_2}, \ldots, A_{i_\ell} \rangle$ von Kartons berechnet, sodass $A_{i_j} \prec A_{i_{j+1}}$ für alle $1 \leq j < \ell$. Die asymptotische Worst-Case-Laufzeit $W(n)$ Ihres Algorithmus soll in $\mathcal{O}(n^i k^j)$ liegen für geeignete Wahlen von $i, j \in \mathbb{N}$. Geben Sie i und j an so dass kein $i' \in \mathbb{N}$ mit $i' < i$ existiert so dass $W(n) \in \mathcal{O}(n^{i'} k^j)$ und so dass kein $j' \in \mathbb{N}$ mit $j' < j$ existiert so dass $W(n) \in \mathcal{O}(n^i k^{j'})$. Begründen Sie, wieso der von Ihnen entwickelte Algorithmus diese Laufzeitschranke einhält.

Lösung:

a) Seien die k-dimensionalen Kartons A, B und C gegeben durch die Abmessungen $(a_1, \ldots, a_k)$, $(b_1, \ldots, b_k)$ bzw. $(c_1, \ldots, c_k)$ so dass $A \overset{(1)}{\prec} B \overset{(2)}{\prec} C$.

Wegen (1) existiert eine Permutation π_{AB} so dass

$$a_{\pi_{AB}(1)} < b_1, a_{\pi_{AB}(2)} < b_2, \ldots, a_{\pi_{AB}(k)} < b_k$$

und wegen (2) existiert eine (möglicherweise andere) Permutation π_{BC} so dass

$$b_{\pi_{BC}(1)} < c_1, b_{\pi_{BC}(2)} < c_2, \ldots, b_{\pi_{BC}(k)} < c_k$$

Sei $\pi_{AC}(i) = (\pi_{AB} \circ \pi_{BC})(i) = \pi_{AB}(\pi_{BC}(i))$ die Permutation, die erst π_{BC} anwendet und anschließend π_{AB}.

Sei $i \in \{1, \ldots, k\}$ beliebig. Dann gilt:

$$a_{\pi_{AC}(i)} = a_{\pi_{AB}(\pi_{BC}(i))} \overset{(1)}{<} b_{\pi_{BC}(i)} \overset{(2)}{<} c_i$$

Daher gilt $A \prec C$.

b) Seien Kartons A und B gegeben. Seien π_A und π_B die Permutationen, die die Abmessungen von A bzw. B sortieren, also so dass

$$a_{\pi_A(1)} < a_{\pi_A(2)} < \ldots a_{\pi_A(k)}$$

und

$$b_{\pi_B(1)} < b_{\pi_B(2)} < \ldots b_{\pi_B(k)}$$

Es existiert genau dann eine Permutation π so dass

$$a_{\pi(1)} < b_1, a_{\pi(2)} < b_2, \ldots, a_{\pi(k)} < b_k$$

wenn

$$a_{\pi_A(1)} < b_{\pi_B(1)}, a_{\pi_A(2)} < b_{\pi_B(2)}, \ldots, a_{\pi_A(k)} < b_{\pi_B(k)}$$

Daher müssen wir die Abmessungen von A und B lediglich in Zeit $\mathcal{O}(2k \log_2 k) = \mathcal{O}(k \log_2 k)$ sortieren und können dann in Zeit $\mathcal{O}(k)$ die Abmessungen komponentenweise vergleichen. Als Gesamtlaufzeit ergibt sich also $\mathcal{O}(k \log_2 k + k) \subset \mathcal{O}(k^2)$

c) Der gesuchte Algorithmus könnte wie folgt vorgehen:

- Konstruiere einen gerichteten, azyklischen, kantengewichteten Graphen $G = (V, E, W)$
- Nehme als Knotenmenge $V = \{A_1, \ldots, A_n\}$. Worst-Case-Laufzeit: $\Theta(n)$
- Iteriere über alle Kartons und sortiere die Abmessungen. Worst-Case-Laufzeit: $\Theta(nk \log_2(k))$
- Prüfe für alle Kartons A_j für welche Kartons A_i gilt, dass $A_i \prec A_j$. Füge für jedes solche Karton-Paar die Kante (A_i, A_j) in den Graphen G ein. Worst-Case-Laufzeit: $\Theta(n^2 k)$, da es im Worst-Case $\Theta(n^2)$ Knotenpaare gibt und der Vergleich von zwei Kartons Worst-Case-Laufzeit $\Theta(k)$ hat
- Setze $W(e) = 1$ für alle $e \in E$. Laufzeit: $\Theta(n^2)$, da es im Worst-Case $\Theta(n^2)$ viele Kanten in G gibt.
- Benutze den Algorithmus aus Aufgabe 7a) um den längsten Pfad (und dessen Länge) in G zu berechnen. Diese Länge ist das gesuchte Ergebnis. Worst-Case-Laufzeit: $\Theta(n + n^2)$, da die Knotenmenge n Elemente besitzt und es maximal n^2 Kanten in G gibt.

Die Worst-Case-Laufzeit des Verfahrens liegt in

$$\Theta(n + nk \log_2(k) + n^2 k + n^2 + (n + n^2)) = \Theta(nk \log_2(k) + n^2 k) \subset \mathcal{O}(n^2 k^2)$$

Die gesuchten i und j sind also $i = 2$ und $j = 2$, da $\mathcal{O}(nk^2) \not\subset \mathcal{O}(nk \log_2(k) + n^2 k)$ und $\mathcal{O}(n^2 k) \not\subset \mathcal{O}(nk \log_2(k) + n^2 k)$.

Aufgabe 9 ($\mathcal{O}$-Notation): (5 Minuten, 3 Punkte)

Beweisen oder widerlegen Sie:

$$\Pi_{i=1}^{n}(i + n) \in \mathcal{O}(n^{2n})$$

Lösung:

Behauptung: Die Aussage gilt.

Beweis:
Zu zeigen: $\exists c \in \mathbb{R}^{>0}.\ \exists n_0 \in \mathbb{N}.\ \forall n \in \mathbb{N}.\ n \geq n_0 \rightarrow (\Pi_{i=1}^{n}(i + n) \leq c \cdot n^{2n})$.
Es gilt für alle $n \geq 2$:

$$\begin{aligned} \Pi_{i=1}^{n}(i + n) &\leq \Pi_{i=1}^{n}(2n) \\ &= (2n)^n \\ &\leq (n^2)^n \\ &= n^{2n} \end{aligned}$$

Wähle also $c = 1$ und $n_0 = 2$. Damit ist die Aussage bewiesen. □

Aufgabe 10 (Rekursionsgleichungen): (10 Minuten, 2 + 2 + 2 = 6 Punkte)

a) Geben Sie für das Programm

```
int max(int a[], int low, int high) {
    if (low >= high) {
        return a[low];
    } else {
        int mid = (low + high) / 2;
        int leftmax = max(a, low, mid);
        int rightmax = max(a, mid + 1, high);
        if (leftmax > rightmax) {
            return leftmax;
        } else {
            return rightmax;
        }
    }
}
```

eine Rekursionsgleichung für die asymptotische Laufzeit des Aufrufes `max(a, 0, n - 1)` an, wobei Sie annehmen dürfen, dass das Array `a` die Länge `n` hat. Die elementaren, also die für die asymptotische Laufzeit relevanten, Operationen sind alle arithmetischen Operationen sowie Vergleiche. Sie brauchen die Basisfälle der Rekursionsgleichung *nicht* anzugeben.

Lösen Sie anschließend die Rekursionsgleichung, indem Sie die asymptotische Komplexitätsklasse (Θ) in geschlossener Form angeben.

b) Geben Sie für das Programm

```
int rev(int n, int m) {
    if (n == 0) {
        return m;
    } else {
        return rev(n - 1, n + 1 + m);
    }
}
```

eine Rekursionsgleichung für die asymptotische Laufzeit des Aufrufes `rev(n, m)` an. Die elementaren, also die für die asymptotische Laufzeit relevanten, Operationen sind alle arithmetischen Operationen sowie Vergleiche. Sie brauchen die Basisfälle der Rekursionsgleichung *nicht* anzugeben.

Lösen Sie anschließend die Rekursionsgleichung, indem Sie die asymptotische Komplexitätsklasse (Θ) in geschlossener Form angeben.

c) Bestimmen Sie für die Rekursionsgleichung

$$T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n^2 + \sqrt{n} + 3 \cdot n + 3$$

die Komplexitätsklasse Θ mit Hilfe des Master-Theorems. Begründen Sie Ihre Antwort.

Lösung:

a) Die Größe der in einem Aufruf zu betrachtenden Eingabe ist $\max(0, \texttt{high} - \texttt{low}) + 1$. Da die Differenz nur im Abbruchfall negativ werden kann, vereinfacht sich dies zu $\texttt{high} - \texttt{low} + 1$. Damit hat die initiale Eingabe die Größe `n`. Die beiden rekursiven Aufrufe haben die Größen $\lfloor \frac{\texttt{n}}{2} \rfloor$ und $\lceil \frac{\texttt{n}}{2} \rceil$.

Insgesamt ergibt sich also für eine Konstante c die Rekursionsgleichung:

$$T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + c$$

Durch Vernachlässigung der Gaußklammern und Rundungen ergibt sich:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + c$$

In geschlossener Form ist die Laufzeit in $\Theta(n)$ (erster Fall des Mastertheorems).

b) Der Kontrollfluss des Algorithmus hängt nur von `n` ab. Der rekursive Aufruf erfolgt mit `n - 1`. Es ergibt sich für eine Konstante c die Rekursionsgleichung:

$$T(n) = T(n-1) + c$$

In geschlossener Form ist die Laufzeit in $\Theta(T(0) + \Sigma_{i=1}^{n} c) = \Theta(n)$.

c) Es sind $b = 9$, $c = 3$ und $f(n) = n^2 + \sqrt{n} + 3 \cdot n + 3$. E wird nun wie folgt bestimmt:

$$E = \frac{\log(9)}{\log(3)} = \frac{2}{1} = 2$$

Damit gilt $n^E = n^2$.

Weiterhin ist $f(n) \in \Theta(n^2)$, denn es gilt:

$$\lim_{n\to\infty} \frac{n^2 + \sqrt{n} + 3 \cdot n + 3}{n^2} = \lim_{n\to\infty} (1 + n^{-1.5} + 3 \cdot n^{-1} + 3 \cdot n^{-2}) = 1$$

Damit gilt der zweite Fall des Mastertheorems und wir erhalten $T(n) \in \Theta(n^2 \cdot \log(n))$.