Prüfungsteilnehmer	Prüfungstermin	Einzelprüfungsnummer

Kennzahl: _____

Kennwort: _____

Arbeitsplatz-Nr.: _____

Frühjahr 2017

66115

Erste Staatsprüfung für ein Lehramt an öffentlichen Schulen

— Prüfungsaufgaben —

Fach: **Informatik (vertieft studiert)**

Einzelprüfung: **Theoret. Informatik, Algorithmen**

Anzahl der gestellten Themen (Aufgaben): **2**

Anzahl der Druckseiten dieser Vorlage: **14**

Bitte wenden!

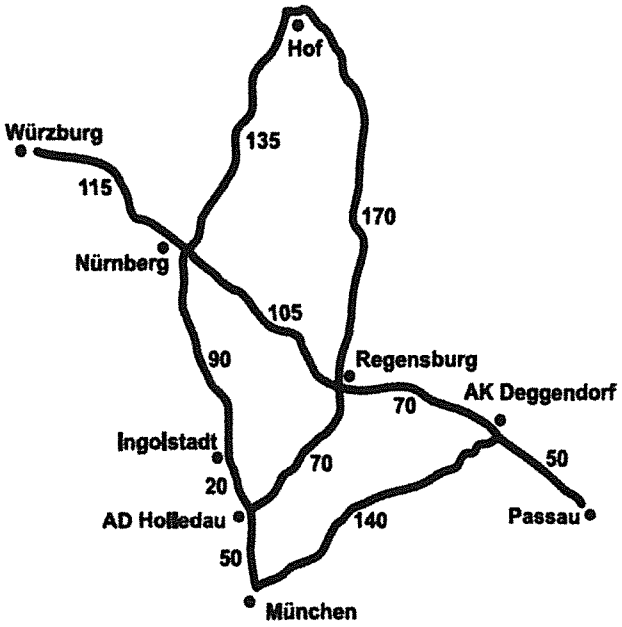
Thema Nr. 1

(Aufgabengruppe)

Es sind alle Aufgaben dieser Aufgabengruppe zu bearbeiten!

Aufgabe 1 (Graphalgorithmen)

Die folgende Abbildung zeigt die wichtigsten bayerischen Autobahnen zusammen mit einigen anliegenden Orten und die Entfernungen zwischen diesen.



Entfernungstabelle

von	nach	km
Würzburg	Nürnberg	115
Nürnberg	Regensburg	105
Regensburg	AK Deggendorf	70
AK Deggendorf	Passau	50
Hof	Nürnberg	135
Nürnberg	Ingolstadt	90
Ingolstadt	AD Holledau	20
AD Holledau	München	50
München	AK Deggendorf	140
Hof	Regensburg	170
Regensburg	AD Holledau	70

a) Bestimmen Sie mit dem Algorithmus von *Dijkstra* den kürzesten Weg von Ingolstadt zu allen anderen Orten. Verwenden Sie zur Lösung eine Tabelle gemäß folgendem Muster und markieren Sie in jeder Zeile den jeweils als nächstes zu betrachtenden Ort. Setzen Sie für die noch zu bearbeitenden Orte eine Prioritäts*warteschlange* ein, d.h. bei gleicher Entfernung wird der ältere Knoten gewählt.

Ingolstadt	Hof	Würzburg	Nürnberg	Regensburg	AK Deggendorf	AD Holledau	Passau	München
0	∞	∞	∞	∞	∞	∞	∞	∞
...								
Ergebnis:								

Fortsetzung nächste Seite!

b) Die bayerische Landesregierung hat beschlossen, die eben betrachteten Orte mit einem breitbandigen Glasfaser-Backbone entlang der Autobahnen zu verbinden. Dabei soll aus Kostengründen so wenig Glasfaser wie möglich verlegt werden. Identifizieren Sie mit dem Algorithmus von Kruskal diejenigen Strecken, entlang welcher Glasfaser verlegt werden muss. Geben Sie die Ortspaare (Autobahnsegmente) in der Reihenfolge an, in der Sie sie in Ihre Verkabelungsliste aufnehmen.

c) Um Touristen den Besuch aller Orte so zu ermöglichen, dass sie dabei jeden Autobahnabschnitt genau einmal befahren müssen, bedarf es zumindest eines sogenannten offenen Eulerzugs. Zwischen welchen zwei Orten würden Sie eine Autobahn bauen, damit das bayerische Autobahnnetz mindestens einen Euler-Pfad enthält?

Fortsetzung nächste Seite!

Aufgabe 2 (Sortierverfahren)

In dieser Aufgabe sei vereinfachend angenommen, dass sich Top-Level-Domains (TLD) ausschließlich aus zwei oder drei der 26 Kleinbuchstaben des deutschen Alphabets ohne Umlaute zusammensetzen. Im Folgenden sollen TLDs lexikographisch aufsteigend sortiert werden, d.h. eine TLD (s_1, s_2) mit zwei Buchstaben (z. B. „co" für Kolumbien) wird also vor einer TLD (t_1, t_2, t_3) der Länge drei (z. B. „com") einsortiert, wenn $s_1 < t_1 \vee (s_1 = t_1 \wedge s_2 \leq t_2)$ gilt.

a) Sortieren Sie zunächst die Reihung [„de", „com", „uk", „org", „co", „net", „fr", „ee"] schrittweise unter Verwendung des *Radix*-Sortierverfahrens (Bucketsort). Erstellen Sie dazu eine Tabelle wie das folgende Muster und tragen Sie dabei in das Feld „Stelle" die Position des Buchstabens ein, nach dem im jeweiligen Durchgang sortiert wird (das Zeichen am TLD-Anfang habe dabei die „Stelle" 1).

Stelle	*Reihung*							
–	de	com	uk	org	co	net	fr	ee
...								

b) Sortieren Sie nun die gleiche Reihung wieder schrittweise, diesmal jedoch unter Verwendung des *Mergesort*-Verfahrens (Sortieren durch Mischen). Erstellen Sie dazu eine Tabelle wie das folgende Muster und vermerken Sie in der ersten Spalte jeweils welche Operation durchgeführt wurde: Wenn Sie die Reihung geteilt haben, schreiben Sie in die linke Spalte ein T und markieren Sie die Stelle, an der Sie die Reihung geteilt haben, mit einem senkrechten Strich „ | ". Wenn Sie zwei Teilreihungen durch Mischen zusammengeführt haben, schreiben Sie ein M in die linke Spalte und unterstreichen Sie die zusammengemischten Einträge. Beginnen Sie mit dem rekursiven Abstieg immer in der linken Hälfte einer (Teil-)Reihung.

Op.	*Reihung*
T	de, com, uk, org | co, net, fr, ee
...	

c) Implementieren Sie das Sortierverfahren *Quicksort* für String-TLDs in einer gängigen Programmiersprache Ihrer Wahl. Ihr Programm (Ihre Methode) wird mit drei Parametern gestartet: dem String-Array mit den zu sortierenden TLDs selbst sowie jeweils der Position des ersten und des letzten zu sortierenden Eintrags im Array.

Fortsetzung nächste Seite!

Aufgabe 3 (Rekursion und Dynamische Programmierung)

Gegeben seien die folgenden Formeln zur Berechnung der *ersten* Fibonacci-Zahlen:

$$fib_n = \begin{cases} 1 & \text{falls } n \leq 2 \\ fib_{n-1} + fib_{n-2} & \text{sonst} \end{cases}$$

sowie der Partialsumme der Fibonacci-Quadrate:

$$sos_n = \begin{cases} fib_n & \text{falls } n = 1 \\ fib_n^2 + sos_{n-1} & \text{sonst} \end{cases}$$

Sie dürfen im Folgenden annehmen, dass die Methoden nur mit $1 \leq n \leq 46$ aufgerufen werden, so dass der Datentyp `long` zur Darstellung aller Werte ausreicht.

a) Implementieren Sie die obigen Formeln zunächst rekursiv (ohne Schleifenkonstrukte wie `for` oder `while`) und ohne weitere Optimierungen („naiv") in Java als:

```
long fibNaive(int n) {
```

bzw.

```
long sosNaive(int n) {
```

b) Offensichtlich ist die naive Umsetzung extrem ineffizient, da viele Zwischenergebnisse wiederholt rekursiv ausgewertet werden müssen. Die Dynamische Programmierung (DP) erlaubt es Ihnen, die Laufzeit auf Kosten des Speicherbedarfs zu reduzieren, indem Sie alle einmal berechneten Zwischenergebnisse speichern und bei erneutem Bedarf „direkt abrufen". Implementieren Sie obige Formeln nun rekursiv aber mittels DP in Java als:

```
long fibDP(int n) {
```

bzw.

```
long sosDP(int n) {
```

c) Am "einfachsten" und bzgl. Laufzeit [in $\mathcal{O}(n)$] sowie Speicherbedarf [in $\mathcal{O}(1)$] am effizientesten ist sicherlich eine iterative Implementierung der beiden Formeln. Geben Sie eine solche in Java an als:

```
long fIter(int n) {
```

bzw.

```
long sosIter(int n) {
```

Fortsetzung nächste Seite!

Aufgabe 4 (Formale Verifikation)

Sie dürfen im Folgenden davon ausgehen, dass keinerlei Under- oder Overflows auftreten (der Datentyp `long` also einen beliebig großen Wertebereich hat).

Gegeben sei folgendes rekursives Programmfragment in der Sprache Java für $n \geq 0$:

```
long sumOfSquares(long n) {
    if (n == 0)
      return 0;
    else
      return n * n + sumOfSquares(n - 1);
}
```

a) Beweisen Sie *formal* mittels *vollständiger Induktion*:

$$\forall n \in \mathbb{N}_0 : \texttt{sumOfSquares(n)} = \frac{n(n+1)(2n+1)}{6}$$

b) Beweisen Sie die Terminierung von `sumOfSquares(n)` für alle $n \geq 0$.

Aufgabe 5 (Aussagen)

Zeigen oder widerlegen Sie die folgenden **Aussagen** (die jeweiligen Beweise sind sehr kurz):

a) Alle regulären Sprachen liegen in NP.

b) Es gibt Sprachen A, B mit $A \subseteq B$, sodass B regulär und A kontextfrei, aber nicht regulär ist.

c) Es gibt unentscheidbare Sprachen L über dem Alphabet Σ, so dass sowohl L als auch das Komplement $\overline{L} = \Sigma^* \setminus L$ rekursiv aufzählbar (= partiell-entscheidbar) sind.

d) Sei L eine beliebige kontextfreie Sprache über dem Alphabet Σ. Dann ist das Komplement $\overline{L} = \Sigma^* \setminus L$ entscheidbar.

Schreiben Sie zuerst zur Aussage „Stimmt" oder „Stimmt nicht" und dann Ihre Begründung.

Fortsetzung nächste Seite!

Aufgabe 6 (Turing)

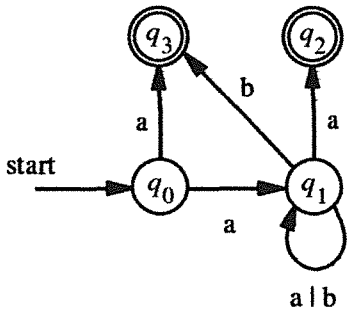
Es sei E die Menge aller (geeignet codierten) Turingmaschinen M mit folgender Eigenschaft: Es gibt eine Eingabe w, so dass M gestartet auf w mindestens 1000 Schritte rechnet und dann irgendwann hält.

Das Halteproblem auf leerer Eingabe H_0 ist definiert als die Menge aller Turingmaschinen, die auf leerer Eingabe gestartet, irgendwann halten.

a) Zeigen Sie, dass E unentscheidbar ist (etwa durch Reduktion vom Halteproblem H_0).
b) Begründen Sie, dass E partiell entscheidbar ist.
c) Geben Sie ein Problem an, welches nicht einmal partiell entscheidbar ist.

Aufgabe 7 (Automaten)

a) Gegeben sei der folgende nichtdeterministische endliche Automat N:



Konstruieren Sie zu N mit Hilfe der Potenzmengen-Konstruktion einen äquivalenten deterministischen endlichen Automaten A. Zeichnen Sie nur die vom Startzustand erreichbaren Zustände ein, die aber *alle*. Die Zustandsnamen von A müssen erkennen lassen, wie sie zustande gekommen sind. Führen Sie *keine* „Vereinfachungen" durch!

Hinweis: In einem deterministischen endlichen Automaten muss es an jedem Zustand für jedes Zeichen einen Übergang geben.

b) Geben Sie einen regulären Ausdruck $\alpha(N)$ für die Sprache, die der nichtdeterministische endliche Automat N aus (a) akzeptiert, an.

c) Sei $L = \{a^k b^\ell \mid k, \ell \in \mathrm{IN}, k > \ell\}$. Jemand behauptet, einen deterministischen endlichen Automaten A mit Zustandsmenge $Q = \{q_0, \ldots, q_{n-1}\}$, Startzustand q_0 und Endzustandsmenge F konstruiert zu haben mit $L = L(A)$.
Geben Sie in Abhängigkeit von A ein Wort $z \in L$ an, das folgende Eigenschaft besitzt: Aus einer akzeptierenden Rechnung von A für z können Sie ein Wort $\hat{z}$ konstruieren mit der Eigenschaft:
(i) A akzeptiert $\hat{z}$ und (ii) $\hat{z} \notin L$.
Beweisen Sie konkret die Eigenschaften (i) und (ii) für Ihr Wort z.

Fortsetzung nächste Seite!

Aufgabe 8 (Halteproblem und *P = NP*)

Sei L die durch den regulären Ausdruck $(10)^* \cup ((101 \cup 11)^* \cup 111)^*$ beschriebene Sprache.
[Alternative Schreibweise: $(10)^* + ((101 + 11)^* + 111)^*$]

a) Sei H_m das Halteproblem bei fester Eingabe m.
Zeigen Sie: L kann auf H_m reduziert werden. (Als Relation: $L \leq H_m$)

b) Angenommen, es wurde gezeigt, dass P = NP ist. Zeigen Sie, dass L dann NP-vollständig ist.

Thema Nr. 2

(Aufgabengruppe)

Es sind alle Aufgaben dieser Aufgabengruppe zu bearbeiten!

Aufgabe 1: Reguläre Sprachen

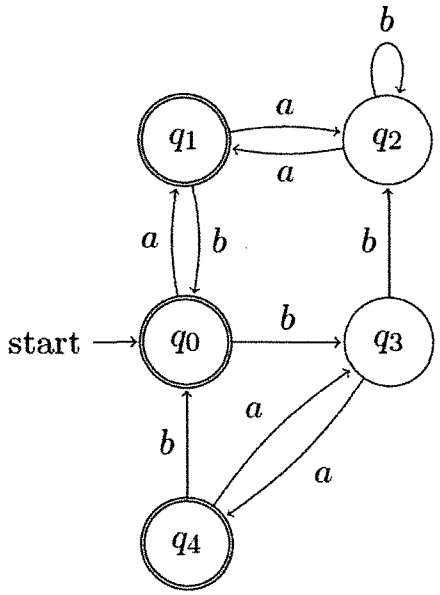
1. Sei $\Sigma = \{a, b\}$ und L_1 die Sprache aller Wörter über Σ, in denen das Wort ab nicht vorkommt. Sei weiterhin L_2 die Sprache aller Wörter über Σ, in denen b genau zwei mal vorkommt.

   a) Geben Sie einen (möglicherweise nichtdeterministischen) endlichen Automaten an, der L_1 akzeptiert.

   b) Geben Sie einen (möglicherweise nichtdeterministischen) endlichen Automaten an, der L_2 akzeptiert.

   c) Geben Sie einen regulären Ausdruck für $L_1 \cap L_2$ an.

2. Gegeben sei der unten aufgeführte deterministische endliche Automat A. Geben Sie einen minimalen deterministischen Automaten für die von A akzeptierte Sprache an.



Fortsetzung nächste Seite!

Aufgabe 2: Kontextfreie Sprachen

1. Gegeben sei die kontextfreie Grammatik $G = (V, \Sigma, P, S)$ mit Sprache $L(G)$, wobei $V = \{S, T, U\}$ und $\Sigma = \{a, b, c, d, e\}$. P bestehe aus den folgenden Produktionen:

$$S \to U \mid SbU \qquad T \to dSe \mid a \qquad U \to T \mid UcT$$

   a) Zeigen Sie $acdae \in L(G)$.
   b) Bringen Sie G in Chomsky-Normalform.

2. Geben Sie eine kontextfreie Grammatik für $L = \{a^i b^k c^i \mid i, k \in \mathbb{N}\}$ an.

3. Zeigen Sie, dass $L = \{a^i b^k c^i \mid i, k \in \mathbb{N} \wedge i < k\}$ nicht kontextfrei ist, indem Sie das Pumping-Lemma für kontextfreie Sprachen anwenden.

Aufgabe 3: Berechen- und Entscheidbarkeit

1. Primitiv rekursive Funktionen
   a) Zeigen Sie, dass die folgendermaßen definierte Funktion $\text{if} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ primitiv rekursiv ist.

$$\text{if}(b, x, y) = \begin{cases} x & \textit{falls } b=0 \\ y & \textit{sonst} \end{cases}$$

   b) Wir nehmen eine primitiv rekursive Funktion $p : \mathbb{N} \to \mathbb{N}$ an und definieren $g(n)$ als die Funktion, welche die größte Zahl $i \leq n$ zurückliefert, für die $p(i) = 0$ gilt. Falls kein solches i existiert, soll $g(n) = 0$ gelten:

$$g(n) = \max(\{i \leq n \mid p(i) = 0\} \cup \{0\})$$

   Zeigen Sie, dass $g : \mathbb{N} \to \mathbb{N}$ primitiv rekursiv ist. (Sie dürfen obige Funktion if als primitiv rekursiv voraussetzen.)

2. Sei $\Sigma = \{a, b, c\}$ und $L \subseteq \Sigma^*$ mit $L = \{a^i b^i c^i \mid i \in \mathbb{N}\}$.
   a) Beschreiben Sie eine Turingmaschine, welche die Sprache L entscheidet. Eine textuelle Beschreibung der Konstruktionsidee ist ausreichend.

   b) Geben Sie Zeit- und Speicherkomplexität (abhängig von der Länge der Eingabe) Ihrer Turingmaschine an.

3. Sei $\Sigma = \{0, 1\}$. Jedes $w \in \Sigma^*$ kodiert eine Turingmaschine M_w. Die von M_w berechnete Funktion bezeichnen wir mit φ_w.

   a) Warum ist $\{w \in \Sigma^* \mid \exists x : \varphi_w(x) = xx\}$ nicht entscheidbar?
   b) Warum ist $\{w \in \Sigma^* \mid \exists x : w = xx\}$ entscheidbar?

Fortsetzung nächste Seite!

Aufgabe 4: Komplexitätstheorie

Eine Menge $U \subseteq V$ heißt *Knotenüberdeckung* eines ungerichteten Graphen $G = (V, E)$, wenn jede Kante des Graphen mit mindestens einem Knoten aus U verbunden ist:

$$\forall (u, v) \in E : u \in U \vee v \in U$$

Das Problem KNOTENÜBERDECKUNG fragt, ob zu einem gegebenen ungerichteten Graphen und einer natürlichen Zahl k eine aus k Knoten bestehende Knotenüberdeckung existiert. Für $G = (V, E)$ und $k \in \mathbb{N}$ gilt also:

$(G, k) \in$ KNOTENÜBERDECKUNG
$\Leftrightarrow$
$\exists U \subseteq V$: U ist Knotenüberdeckung von G und $|U| = k$

1. Begründen Sie, dass KNOTENÜBERDECKUNG in NP liegt.

Eine Menge $C \subseteq V$ heißt *Clique* eines ungerichteten Graphen $G = (V, E)$, wenn alle Paare verschiedener Knoten der Clique durch eine Kante des Graphen verbunden sind:

$$\forall u \in C \;\; \forall v \in C : u \neq v \Rightarrow (u, v) \in E$$

Das Problem CLIQUE fragt, ob zu einem gegebenen ungerichteten Graphen und einer natürlichen Zahl k eine aus k Knoten bestehende Clique existiert. Für $G = (V, E)$ und $k \in \mathbb{N}$ gilt also:

$(G, k) \in$ CLIQUE $\Leftrightarrow \exists C \subseteq V : C$ ist Clique und $|C| = k$

Wir definieren $\overline{E} := \{(u, v) \in V \times V \mid (u, v) \notin E\}$.

2. Zeigen Sie: C ist Clique von $G = (V, E)$, genau dann wenn $V \setminus C$ Knotenüberdeckung von $(V, \overline{E})$ ist.

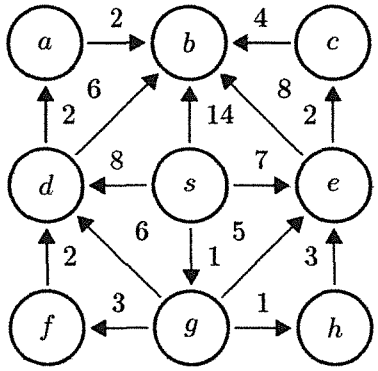
3. Warum ist $r : ((V, E), k) \mapsto ((V, \overline{E}), |V| - k)$ eine polynomielle Reduktion von CLIQUE auf KNOTENÜBERDECKUNG?

Fortsetzung nächste Seite!

Aufgabe 5: Algorithmen und Datenstrukturen I

Führen Sie den Algorithmus von Dijkstra mit Startknoten s auf dem folgenden Graphen durch, um einen Kürzeste-Wege-Baum zu finden. Übernehmen Sie dazu die Berechnungstabelle auf Ihr Lösungsblatt und füllen Sie dort die Zeilen der Schritte 2. bis 9. aus. Markieren Sie zum Schluss alle Kanten, die zum berechneten Kürzeste-Wege-Baum gehören.

Anmerkung: Für den i-ten Schritt enthält die Spalte *Aktueller Knoten* den im i-ten Schritt betrachteten aktuellen Knoten zusammen mit seinem Distanzwert. $(s, 0)$ im ersten Schritt bedeutet also, dass der Knoten s die Distanz 0 zu s hat. Für den i-ten Schritt enthält die Spalte Inhalt der Priority-Queue Paare bestehend aus Knoten und zugehöriger Priorität. $(g, 1)$ bedeutet also beispielsweise, dass Knoten g mit Priorität 1 in der Queue ist.



Schritt	Aktueller Knoten	Vorgänger-Array a	b	c	d	e	f	g	h	Inhalt der Priority-Queue
1.	$(s, 0)$		s		s	s		s		$(g, 1)$, $(e, 7)$, $(d, 8)$, $(b, 14)$
2.										
3.										
4.										
5.										
6.										
7.										
8.										
9.										

Fortsetzung nächste Seite!

Aufgabe 6: Algorithmen und Datenstrukturen II

Für ein Array A bezeichne $A[i]$ das i-te Element von A. Die Elemente eines Arrays der Länge n haben die Indizes 1 bis n.
Das Maximum Subarray Sum Problem (kurz: MSAS Problem) ist wie folgt definiert:

Gegeben: Ein nichtleeres Array A der Länge $n \in \mathbb{N}$ von ganzen Zahlen (d.h. Zahlen aus $\mathbb{Z}$).

Aufgabe: Finden Sie die größte Zahl $s \in \mathbb{Z}$, sodass s die Summe der Einträge eines nichtleeren Teilarrays von A ist. D.h., finden Sie $s = \max\left\{\Sigma_{k=i}^{j} A[k]; 1 \le i \le j \le n\right\}$.

1. Betrachten Sie Algorithmus 1.

Algorithmus 1 : MSAS (A, i, j)

```
1 ℓ ← j − i + 1
2 if ℓ = 1 then
3 |  return A[i]
4 sA ← 0
5 for r ← i to j do
6 |  sA ← sA + A[r]
7 sLinks ← MSAS (A, i, j − 1)
8 sRechts ← MSAS(A, i + 1, j)
9 return max {sA, sLinks, sRechts}
```

a) Begründen Sie weshalb dieser Algorithmus mit dem Aufruf MSAS (A, 1, n) das MSAS Problem löst.
b) Analysieren Sie die Laufzeit des Algorithmus.

2. Algorithmus 1 ist relativ ineffizient, da einige Teilarrays zu oft untersucht werden. Betrachten Sie Algorithmus 2.

   Hierbei ist B ein zweidimensionales *globales* Array, dessen Einträge mit $-\infty$ vorinitialisiert sind. Analysieren Sie die Laufzeit dieses Algorithmus.

Algorithmus 2 : MSAS (A, i, j)

```
1 if B[i][j] ≠ −∞ then
2 |  return B[i][j]
3 ℓ ← j − i + 1
4 if ℓ = 1 then
5 return A[i]
6 sA ← 0
7 for r ← i to j do
8 |  sA ← sA + A[r]
9 sLinks ← MSAS (A, i, j − 1)
10 sRechts ← MSAS (A, i + 1, j)
11 B[i][j] ← max {sA, sLinks, sRechts}
12 return B[i][j]
```

Fortsetzung nächste Seite!

3. Eine bessere Idee, als alle Teilarrays zu betrachten, ist die folgende. Für ein Array B bezeichne s(B) die größte Zahl, die die Summe eines nichtleeren Teilarrays von B ist. Außerdem bezeichne sRechts(B) die größte Zahl, die die Summe eines nichtleeren Teilarrays von B ist, das das letzte Element von B enthält. Sei A_i das Teilarray von A, das aus den Elementen $A[1],..., A[i]$ gebildet wird.

   Angenommen, wir kennen für ein $i \in \{1,\ldots, n\}$ bereits die Zahlen s(A_i) und sRechts(A_i). Dann können wir auf sehr einfache Weise die Zahlen s(A_{i+1}) und sRechts(A_{i+1}) bestimmen.

   Geben Sie einen Algorithmus in Pseudo-Code an, der keine rekursiven Aufrufe verwendet und mit obiger Idee das MSAS Problem löst. Der Algorithmus soll eine möglichst gute Laufzeit besitzen. (Lineare Laufzeit ist möglich.)