Karlsruher Institut für Technologie

Algorithmen 1 – Wintersemester 2018/19
Institut für Theoretische Informatik
`http://verialg.iti.kit.edu`
Prof. Dr. Carsten Sinz – Markus Iser

Nachklausur **08.04.2019**

Name, Vorname:

Matrikelnummer:

Allgemeine Hinweise

- Als Hilfsmittel ist nur *eine* DIN-A4-Seite mit Ihren *handschriftlichen* Notizen zugelassen.
- Schreiben Sie auf *alle* Blätter Ihren Namen und Ihre Matrikelnummer.
- Die durch die Übungsblätter gewonnenen Bonuspunkte werden erst nach Erreichen der zum Bestehen der Klausur nötigen Punktzahl hinzugezählt. Die Anzahl der Bonuspunkte entscheidet nicht über das Bestehen.
- **Die Klausur nur mit Erlaubnis umdrehen!**

Übersicht Punkteverteilung

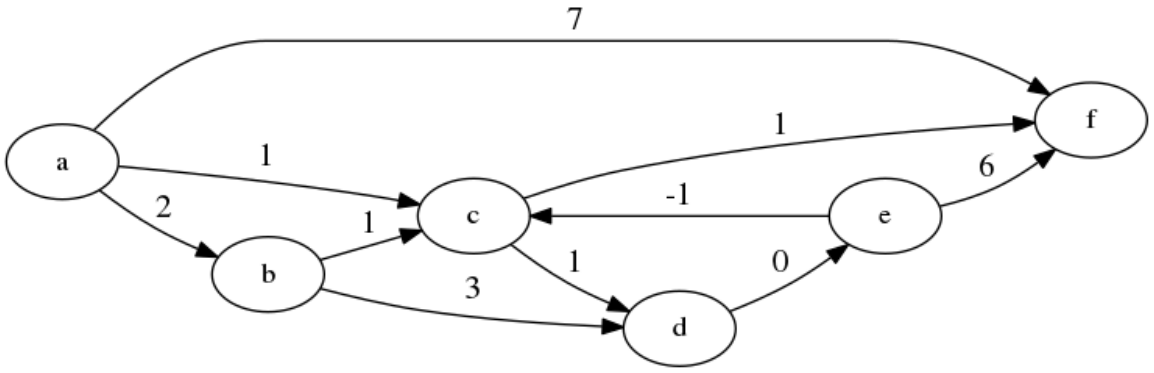
Die Klausur besteht aus drei Teilen, in denen Sie jeweils 20 Punkte erreichen können.

Aufgaben	Teil A							Teil B				Teil C		
	1	2	3	4	5	6	7	1	2	3	4	1	2	3
60 Punkte	20							20				20		
	2	4	2	3	4	3	2	7	5	4	4	7	5	8

A Gemischte Kleinaufgaben (20 Punkte)

A.1 Kürzeste Wege (2 Punkte)

Wieviele kürzeste Wege von a nach f enthält der folgende gerichtete gewichtete Graph? Begründen Sie kurz.



Antwort. Es gibt *unendlich viele* kürzeste Wege (1 Punkt),
wegen einem *Kreis der Länge null* (0.5 Punkte)
auf dem kürzesten Weg von a über c nach f hat $c \to d \to e \to c$ Länge 0 (0.5 Punkte)

A.2 O-Kalkül (4 Punkte)

A.2.1 Teil Eins

Zeigen Sie oder widerlegen Sie, dass $1.5^n \in \Omega(1.5^{2n})$ gilt.

Antwort. Die Behauptung gilt nicht (1 Punkt),
denn $\nexists c > 0, \forall n > n_0, 1.5^n \geq c \cdot 1.5^{2n}$, denn

$\lim_{n\to\infty} \frac{1.5^n}{1.5^{2n}} = \lim_{n\to\infty} \frac{1}{1.5^n} = 0$. (1 Punkt)

A.2.2 Teil Zwei

Zeigen Sie oder widerlegen Sie, dass $7^{\log_5 n} \in O(n^2)$ gilt.

Antwort. Die Behauptung gilt (1 Punkt),
denn $7^{\log_5 n} = 7^{\frac{\log_7 n}{\log_7 5}} = n^{\frac{1}{\log_7 5}} = n^{\log_5 7}$ und $\log_5 7 < 2$. (1 Punkt)

A.3 Union-Find Datenstruktur (2 Punkte)

Gegeben sei das Parent Array in unten stehender Tabelle aus dem *Basis*-Algorithmus Union-Find der Vorlesung.[1] Geben Sie in den freigelassenen Zeilen eine Folge von Union Operationen an, so dass das gegebene Parent Array erzeugt wird. Geben Sie in der ersten Spalte zu jeder Union Operation auch die link Operation an, die ausgeführt wird, und zeigen Sie in den Spalten danach, wie sich das Parent-Array dadurch ändert.

v	1	2	3	4	5	6
union(6,1), link(6,1)	1	2	3	4	5	1
union(3,2), link(3,2)	1	2	2	4	5	1
union(6,4), link(1,4)	4	2	2	4	5	1
parent[v]	4	2	2	4	5	1

A.4 Adjazenzfelddarstellung (3 Punkte)

Gegeben ist folgender Graph in Adjazenzfelddarstellung (mit Knotenfeld V und Kantenfeld E). Übertragen Sei den Graphen in seine Darstellung als Adjazenzmatrix. Tragen Sie diese in das leere Raster unten ein.

V:

0	1	2	3	4	5
0	1	4	5	7	9

E:

0	1	2	3	4	5	6	7	8	9	10
1	0	2	3	4	0	2	1	5	0	3

	0	1	2	3	4	5
0		X				
1	X		X	X		
2					X	
3	X		X			
4		X				X
5	X			X		

[1] *kein* Union-by-Rank, *ohne* Pfadkompression

A.5 Laufzeitkomplexität (4 Punkte)

Tragen Sie die folgenden Algorithmen und Datenstruktur-Operationen aus der Vorlesung entsprechend ihrer asymptotischen *worst-case* Zeitkomplexität (*nicht* amortisiert) in die Felder ein. Tragen Sie jeden Algorithmus genau einmal bei der kleinstmöglichen Laufzeit ein.

Algorithmus	
`binarySearch()`	Binäre Suche in einem Array der Größe n
`shortestPath()`	Dijkstras kürzeste Wege Algorithmus mit binärem Heap auf einem ungerichteten Graphen mit n Knoten und $8n$ Kanten
`merge()`	Zusammenführen von zwei sortierten Listen der Länge n und $2n$ in eine neue sortierte Liste
`last()`	Zugriff auf das letzte Element einer doppelt verketteten Liste der Länge n
`findComponents()`	Bestimmung aller Zusammenhangskomponenten in einem ungerichteten Graph mit $2n$ Knoten und n Kanten
`siftDown()`	Einsieben eines Elements in einen Binary Heap mit n Elementen
`quickSort()`	Quicksort in der besten Variante aus der Vorlesung auf einem Array der Länge n
`insert()`	Einfügen des n-ten Elements in eine Hashtabelle mit linearer Suche

$O(1)$	`last`
$O(\log n)$	`binarySearch`, `siftDown`
$O(n)$	`merge`, `insert`, `findComponents`
$O(n \log n)$	`shortestPath`
$O(n^2)$	`quickSort`

A.6 Mastertheorem (3 Punkte)

Gegeben seien zwei rekursive Algorithmen A und B, wobei B in jedem Rekursionsschritt nicht nur sich selbst, sondern auch A aufruft. Die Laufzeiten von A und B seien durch die Rekurrenzen $T_A(n)$ und $T_B(n)$ beschrieben mit $n = 3^k$ und $k \in \mathbb{N}_{>0}$.

$$T_A(n) = 5n + aT_A(\frac{n}{3}), \qquad T_A(1) = 4$$
$$T_B(n) = T_A(n) + bT_B(\frac{n}{3}), \qquad T_B(1) = 2$$

Geben Sie Werte für a und b aus $\mathbb{N}_{>0}$ an, so dass die Laufzeit von B in $\Theta(n \log n)$ liegt für $n = 3k$. Begründen Sie Ihre Wahl von a und b kurz.

Antwort. Mit $a = 2$ ($a < 3$, 1 Punkt) ist nach Mastertheorem $T_A \in \Theta(n)$ (1 Punkt), so dass dann mit $b = 3$ (1 Punkt) gilt, dass $T_B \in \Theta(n \log n)$.

A.7 Sortieren (2 Punkte)

Geben Sie in Θ-Notation an, welches Worst-Case-Laufzeitverhalten ein vergleichsbasierter Sortieralgorithmus bestenfalls haben kann. Nennen Sie ein Beispiel für einen vergleichsbasierten Sortieralgorithmus, der in diesem Sinne optimal und zusätzlich stabil ist.

$\Theta(n \log n)$ (1 Punkt),
Mergesort (1 Punkt)

B Algorithmen Ausführen (20 Punkte)

B.1 Starke Zusammenhangskomponenten (7 Punkte)

Der Algorithmus von Kosaraju findet in einem gerichteten Graphen $G = (V, E)$ alle starken Zusammenhangskomponenten. Eine starke Zusammenhangskomponente ist ein maximaler Teilgraph $H = (W, F)$ von G (mit $W \subseteq V$ und $F \subseteq E$), in dem es für jedes Knotenpaar $(u, v) \in W \times W$ einen Pfad von u nach v und von v nach u gibt.

Algorithmus 1 : Algorithmus von Kosaraju

```
Input : Graph G = (V, E)
1  for u ∈ V do
2  |  R[u] ← ⊥
3  |  mark u as unvisited
4  Stack S = ∅
5  for u ∈ V do
6  |  visit(u)
7  while S ≠ ∅ do
8  |  u ← pop(S)
9  |  assign(u, u)
10 return R
```

Subroutine 1 : `visit`

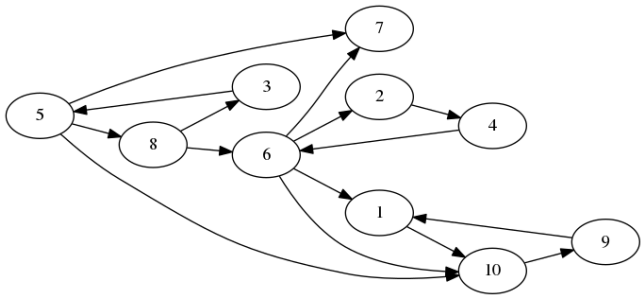
```
Input : Graph G = (V, E), Vertex u
1  if u is unvisited then
2  |  mark u as visited
3  |  for x ∈ V with (x, u) ∈ E do
4  |  |  visit(x)
5  |  push(S, u)
```

Subroutine 2 : `assign`

```
Input : Graph G = (V, E), Vertex u, Vertex r
1  if R[u] = ⊥ then
2  |  R[u] ← r
3  |  for x ∈ V with (u, x) ∈ E do
4  |  |  assign(x, r).
```

Algorithmus 1 beginnt (in den Zeilen 1 bis 4) mit einigen Initialisierungsschritten. In den Zeilen 5 und 6 werden dann zunächst die Knoten *entgegen der Pfeilrichtung* (Subroutine 1, Zeile 3) rekursiv abgelaufen und entsprechend der Reihenfolge des Austritts aus der Rekursion auf den globalen Stack S gelegt (Subroutine 1, Zeile 5).

Anschließen werden (in den Zeilen 7 bis 9) die Knoten in der Reihenfolge, in der sie auf dem globalen Stack S liegen besucht. Hierbei wird jedem Knoten rekursiv ein Wurzelknoten r zugewiesen (Subroutine 2, Zeile 2). Dieser Wurzelknoten wird im global Array R gespeichert und steht am Ende stellvertretend für eine Zusammenhangskomponente.



B.1.1 Algorithmus ausführen (3 + 2 Punkte)

Führen Sie den Algorithmus von Kosaraju auf dem oben angegebenen Graphen aus. Halten Sie sich bei allen Iterationen an die von den Knotennummern induzierte Reihenfolge, und beginnen Sie immer mit dem kleinsten Knoten.

Stack ausgeben. Geben Sie vor der Ausführung von Zeile 7 den Zustand des Stack S aus.

S:	2	4	3	5	8	6	10	9	1	7

Wurzelknoten ausgeben. Geben Sie in Zeile 10 den Array R der Wurzelknoten aus.

R:	1	2	3	4	5	6	7	8	9	10
	1	6	8	6	8	6	7	8	1	1

B.1.2 Zusatzfrage (1 Punkt)

Wie heißt die Datenstruktur aus der Vorlesung, bei der (wie hier bei `assign`) mit Repräsentanten gearbeitet wird?

Antwort. union-find

B.1.3 Zusatzfrage (1 Punkt)

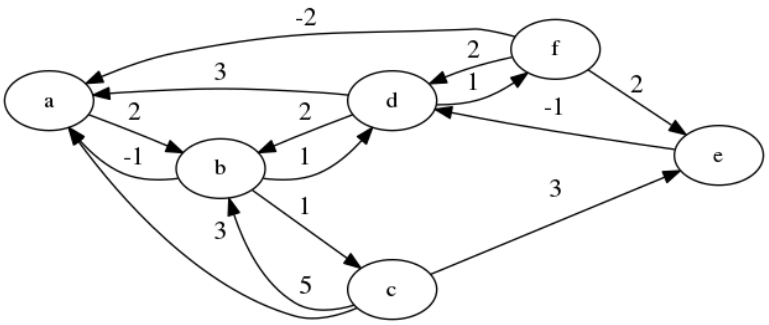
Wird in `assign` und `visit` Tiefen- oder Breitensuche benutzt?

Antwort. Tiefensuche

B.2 Algorithmus von Bellman-Ford (3+2 Punkte)

Führen Sie auf dem unten gegebenen gerichteten Graphen mit Kantengewichten den Bellman-Ford-Algorithmus mit Startknoten c aus.

Rechnen. Tragen Sie die kürzeste Distanz zwischen c und jedem Knoten in die Tabelle ein.



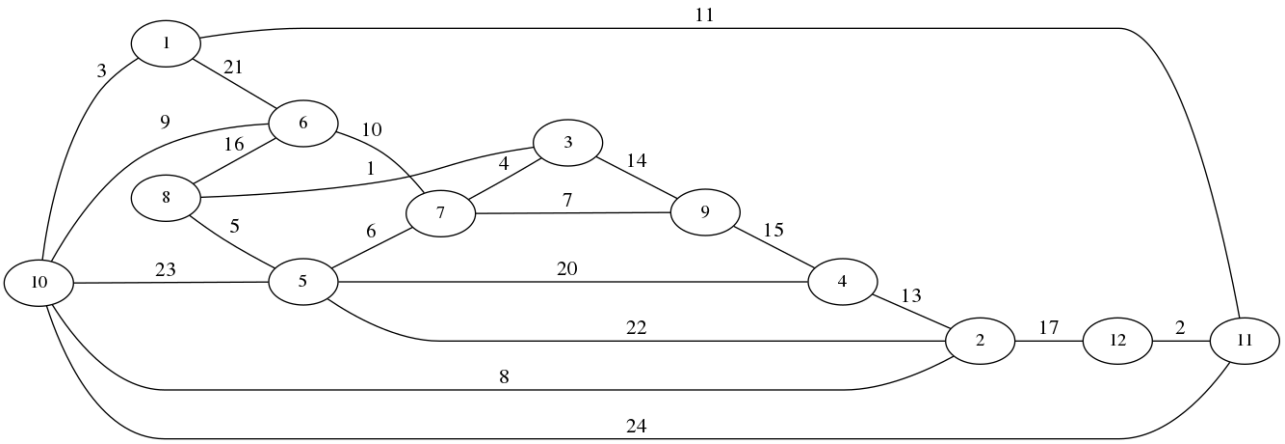
a	b	c	d	e	f
1	3	0	2	3	3

Zeichnen. Zeichnen Sie den vom Bellman-Ford-Algorithmus berechneten Baum kürzester Wege:

$c \to e \to d \to f \to a \to b$

B.3 Algorithmus von Kruskal (4 Punkte)

Berechnen Sie vom unten gegebenen ungerichteten gewichteten Graphen einen Minimum Spanning Tree (MST) mit dem Algorithmus von Kruskal. Tragen Sie in die Tabelle für jede Kante e eine Zahl x ein, die angibt, als wievielte Kante e zum MST hinzugefügt wurde oder tragen Sie "$-$" ein, falls e nicht zum MST gehört.



Kante	$(1,6)$	$(1,10)$	$(1,11)$	$(2,4)$	$(2,5)$	$(2,10)$	$(2,12)$	$(3,7)$	$(3,8)$	$(3,9)$	$(4,5)$
Zahl	–	3	10	11	–	7	–	4	1	–	–

Kante	$(4,9)$	$(5,7)$	$(5,8)$	$(5,10)$	$(6,7)$	$(6,8)$	$(6,10)$	$(7,9)$	$(11,12)$	$(10,11)$	
Zahl	–	–	5	–	9	–	8	6	2	–	

B.4 Hashing (4 Punkte)

Wir betrachten in dieser Aufgabe Hashtabellen mit m Stellen. Als Hashfunktion verwenden wir die Funktion $h_m(x) = x \mod m$. Zur Kollisionsauflösung wird lineare Suche angewendet. Folgende Hashtabelle hat die Größe $m = 10$.

0	1	2	3	4	5	6	7	8	9
20	51	10	32		55	76	106		19

B.4.1 Folge von Insert Operationen (3 Punkte)

Sei eine leere Hashtabelle mit $m = 10$ und Hashfunktion h_{10} gegeben. Geben Sie eine Folge von insert-Operationen an, so dass die Tabelle nach Ausführen dieser Operationsfolge den obigen Zustand hat.

Antwort. Folgende Reihenfolge gilt (Zeilen können vertauscht sein):

- insert 20, insert 51, insert 10, insert 32,
- insert 55,
- insert 76, insert 106,
- insert 19

B.4.2 Lesezugriffe (1 Punkt)

Wie viele Lesezugriffe auf belegte Felder im Array der obigen Hashtabelle werden bei Ihrer Operationsfolge ausgeführt?

Antwort. 4 oder 12 (gilt beides)

C Algorithmenentwurf (20 Punkte)

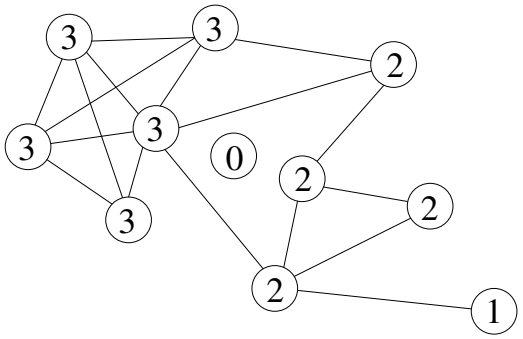
C.1 Der k-Core von Graphen (7 Punkte)

Der k-Core eines ungerichteten Graphen $G = (V, E)$ ist gegeben durch die größte Teilmenge $V' \subseteq V$, so dass alle Knoten im knoteninduzierten Teilgraphen $G' := G[V']$ einen Knotengrad $deg_{G'}(v)$ größer gleich k haben.[2]

Die Core-Struktur $C : V \to N_{\geq 0}$ gibt für jeden Knoten v die größte Nummer x an, so dass v noch im x-Core enthalten ist.

C.1.1 Problemverständnis (2 Punkte)

Geben Sie für den Graphen rechts die Core-Struktur C an. Schreiben Sie dazu in jeden Knoten v den Wert $C(v)$.



C.1.2 Algorithmus (5 Punkte)

Gegeben sei ein ungerichteter Graph $G = (V, E)$. Geben Sie einen Algorithmus an, der die Core-Struktur C von G berechnet und ausgibt.

Verwenden Sie in Ihrem Algorithmus die Datenstruktur `PriorityQueue`, welche Knoten nach ihrem Knotengewicht in einem min-Heap organisiert und zusätzlich die Funktion `decreaseKey` anbietet.

Funktion	**Beschreibung**	**Komplexität**
`insert`(v, w)	Fügt Knoten v mit Gewicht w in den Heap ein.	$O(\log n)$
`decreaseKey`(v, w)	Ändert v's Gewicht vom aktuellen Wert w' auf w und stellt die Heap-Eigenschaft wieder her. Vorraussetung ist, dass $w \leq w'$.	$O(\log n)$
`extractMin`	Gibt das Element v mit dem kleinsten Gewicht w zurück.	$O(1)$

Datenstruktur `PriorityQueue`

Ihr Algorithmus soll in $O((|V| + |E|) \log |V|)$ laufen. Begründen Sie das Laufzeitverhalten ihres Algorithmus.

[2] $G[V'] = (V', E' := \{\{u, v\} \in E \mid u, v \in V'\})$

```
   // Array für den Knotengrad, mit 0 initialisiert
 1 deg : Array[1..n] of N;
   // Array für aktive Knoten, mit 1 initialisiert
 2 active : Array[1..n] of 0, 1;
   // Array für die Core-Struktur, mit 0 initialisiert
 3 C : Array[1..n] of N;
 4 Q : Addressable Priority Queue;
 5 for {v, u} ∈ E do
 6 |   deg(u) += 1;
 7 |   deg(v) += 1;
 8 for v ∈ V do
 9 |   Q.insert(v, deg(v));
10 while !Q.empty() do
11 |   v := Q.extractMin();
12 |   C [v] := deg(v);
13 |   active(v) := 0;
14 |   for {v, u} ∈ E and active[u] do
   |   |   /* Knotengrad der Nachbarn updaten                    */
15 |   |   deg[u] -= 1;
16 |   |   Q.decreaseKey(u, deg[u]);
17 return C;
```

Antwort. Der dominierende Faktor in dem Algorithmus ist die letzte `While`-Schleife. Die asymptotische Laufzeit ergibt sich, da jeder Knoten genau einmal aus der Prioritätsliste mit `extractMin` geholt wird und für jede Kante höchstens einmal `decreaseKey` aufgerufen wird.

C.2 Urlaubsdatenbank (5 Punkte)

In einem Unternehmen hat jeder Mitarbeiter eine eindeutige ID $\iota \in \mathbb{N}$. Gegeben sei die Datei D als eine Folge von n Paaren der Form $(\iota, d) \in \mathbb{N} \times \{1, \ldots, 366\}$. Die Datei D beschreibt für ein bestimmtes Jahr, an welchen Tagen die Mitarbeiter Urlaub genommen haben. Für jeden Tag, an dem ein Mitarbeiter in dem Jahr Urlaub genommen hat, enthält D genau ein entsprechendes Paar. Hinweis: Beachten Sie, dass die IDs der Mitarbeiter beliebig groß sein können.

C.2.1 Urlaubstage Zählen (3 Punkte)

Geben Sie einen Algorithmus an, der in erwarteter Zeit $O(n)$ die IDs aller Mitarbeiter in D ausgibt, die in der ersten Hälfte des Jahres weniger als 10 Tage Urlaub genommen haben. Jede ID darf dabei nicht mehr als einmal ausgegeben werden. Der Speicherbedarf Ihres Algorithmus soll ebenfalls in $O(n)$ liegen.

Begründen Sie kurz, warum Ihr Algorithmus das gewünschte Laufzeitverhalten aufweist.

```
1  n ← |D| // O(n)
2  Initialisiere Hashtabelle H : N → N × N mit 2n Slots und verketteten Listen zur Kollisionsauflösung // O(n)
3  for (x, d) ∈ D do
4      if d < 366/2 then
           /* Test x ∈ H ist in erwartet O(1) :                                   */
5          if x ∉ H then
6              insert (x, 1) in H // O(1)
7          else
8              H[x] ← H[x] + 1 // O(1)
   /* O(n) :                                                                     */
9  for (x, c) ∈ H do
10     if c < 10 then
11         gib x aus;
```

C.2.2 Mitarbeiter Zählen (3 Punkte)

Geben Sie einen Algorithmus an, der an Hand von D in deterministisch $O(n)$ Zeit einen Tag des Jahres ermittelt, an dem eine maximale Anzahl Mitarbeiter Urlaub genommen hat.

Begründen Sie kurz, warum Ihr Algorithmus das gewünschte Laufzeitverhalten aufweist.

```
1 Initialisiere Array A[1..366] Z mit 0 // O(n)
2 for (x, d) ∈ D do
3     A[d] ← A[d] + 1;
4 d_max ← 0;
5 for d ∈ [1 ... 366] do
6     if A[d] > A[d_max] then
7         d_max ← d;
8 Gib d_max aus;
```

Allozieren und Initialisieren von A kostet $O(1)$ Zeit, da 366 konstant ist. Durchlaufen von D kostet $O(n)$ Zeit. Durchlaufen von A kostet wieder $O(1)$ Zeit. Insgesamt wird $O(n)$ Zeit benötigt.

C.3 Optimale Zeilenumbrüche im Textsatz (8 Punkte)

Gute automatische Textsatzssysteme bestimmen die Zeilenumbrüche innerhalb eines Absatzes nicht Zeile für Zeile, sondern ziehen auch den Effekt eines Umbruchs auf die darauffolgenden Zeilen in Betracht.

In dieser Aufgabe soll ein Algorithmus basierend auf dynamischer Programmierung entwickelt werden, der optimale Zeilenumbrüche für einen Absatz eines Textes bestimmt. Dabei gehen wir von einer nicht-proportionalen Schriftart aus, d.h. alle Zeichen sind gleich breit, Zeilenumbrüche finden immmer am Ende eines Wortes statt.

Die Eingabe des Algorithmus besteht aus den n Worten $w_1, \ldots, w_n$ eines Absatzes. Die Länge in Zeichen eines Wortes w_i bezeichnen wir mit $|w_i|$. Der Text soll auf Zeilen von jeweils M Zeichen Länge gesetzt werden.

Wenn eine Zeile die Worte w_i bis w_j enthält ($1 \leq i \leq j \leq n$), wobei wir genau ein Leerzeichen zwischen zwei Worte drucken, können sich überschüssige Leerzeichen am Ende einer Zeile z ergeben. Deren Anzahl bezeichnen wir mit L_z und definieren die Kosten (oder *badness*) einer gesetzen Zeile als L_z^2. Passen die Worte w_i bis w_j nicht auf eine Zeile, so sollen die Kosten unendlich sein.

Für einen optimalen Textsatz soll die Summe der Kosten über alle Zeilen (also auch die letzte) minimal sein.

Die Ausgabe des Algorithmus besteht aus einem Array von Indizes i derjenigen Worte w_i, nach denen ein Zeilenumbruch erfolgt.

C.3.1 Berechnung der Kosten für eine Zeile (1 Punkt)

Die Kosten für eine Zeile mit den Worten w_i bis w_j wollen wir mit $w(i, j)$ bezeichnen. Geben Sie eine Formel an, wie sich die Kosten $w(i, j)$ aus den Wortlängen und der Zeilenbreite M ergeben.

Antwort.

$$L_z(i,j) = M - (j-i) - \sum_{k=i}^{j} |w_k|$$

$$w(i,j) = \begin{cases} (L_z(i,j))^2 & \text{falls } L_z(i,j) \geq 0 \\ \infty & \text{sonst} \end{cases}$$

C.3.2 Beispielrechnung $w(i, j)$ (1 Punkt)

Betrachten Sie den aus 5 Worten bestehenden Text

```
Morgenstund hat Gold im Mund
```

und tragen Sie die Werte $w(i, j)$ für $1 \leq i \leq j \leq 5$ für Textbreite $M = 11$ in folgende Tabelle ein:

$i \setminus j$	1	2	3	4	5
1	0	∞	∞	∞	∞
2		64	9	0	∞
3			49	16	∞
4				81	16
5					49

C.3.3 Vorüberlegungen Algorithmus (3 Punkte)

Der Algorithmus soll die Optimierung anhand von Ergebnissen aus Teilproblemen berechnen, die in zwei Feldern (Arrays) der Größe n abgelegt werden:

1. $A[i]$: Die minimalen Kosten ab Wort i bis zum Ende des Textes. (D.h. die minimalen Kosten, wenn der Text mit w_i beginnen würde.)
2. $B[i]$: Der Index $j \geq i$ des Wortes w_j, nach dem der nächste Zeilenumbruch für das optimale Ergebnis erfolgen muss, wenn der Text mit w_i beginnen würde.

Geben Sie (rekursive) Definitionen zur Berechnung der Felder A und B unter Verwendung von $w(i, j)$ an.

$$A[i] = \min_{\substack{j \\ i \leq j \leq n}} \{w(i, j) + A[j + 1]\}$$

$$B[i] = \arg\min_{\substack{j \\ i \le j \le n}} \{w(i,j) + A[j+1]\}$$

Dabei soll $A[n+1] = 0$ sein.

C.3.4 Berechnung (2 Punkte)

Geben Sie einen Algorithmus zur Berechnung der Felder A und B an.

Input : Array $w : [1..n][1..n]$ of Int $\cup \{\infty\}$
Output : Arrays $A[1..n]$, $B[1..n]$

```
1  for i = n down to 1 do
2  |  min = w[i, n]
3  |  mIdx = n
4  |  for j = n − 1 down to i do
5  |  |  a = w[i, j] + A[j + 1]
6  |  |  if a < min then
7  |  |  |  min = a
8  |  |  |  mIdx = j
9  |  |  end
10 |  end
11 |  A[i] = min
12 |  B[i] = mIdx
13 end
14 return A, B
```

C.3.5 Ausgabe (1 Punkt)

Verwenden Sie das Zwischenergebnis B, um die geforderte Ausgabe zu berechnen.

Input : Array $B[1..n]$ of Int
Output : Array $R[1..]$ of Int

```
1 i = 1, p = 1
2 while i ≤ n do
3 |  R[p] = B[i]
4 |  i = B[i] + 1
5 |  p = p + 1
6 end
7 return R
```