Karlsruher Institut für Technologie

Institut für Theoretische Informatik
T.T.-Prof. Dr. Thomas Bläsius
https://scale.iti.kit.edu

Zusatzaufgaben 05

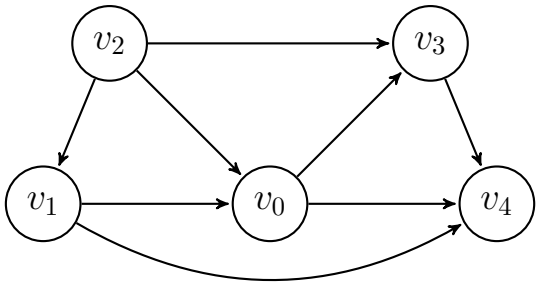
Algorithmen I – Sommersemester 2022

Gesamtpunkte: 35

Aufgabe 1 - WHY? (3 Punkte)

Gegeben sei ein gerichteter Graph $G = (V, E)$. Wir wollen „Y“-Muster in G finden. Ein „Y“-Muster wird dabei gebildet von drei Kanten $(u, v), (w, v), (v, x)$ mit $u, v, w, x \in V$.

1. Gib drei Kanten an, die im folgenden Graphen ein „Y“-Muster bilden (1 Punkt)



2. Beschreibe einen Algorithmus, der in $\Theta(n + m)$ entweder drei Kanten bestimmt, die ein „Y“-Muster bilden, oder angibt, dass kein „Y“-Muster im gegebenen Graphen existiert. Begründe kurz Laufzeit und Korrektheit deines Algorithmus. (2 Punkte)

Lösung 1

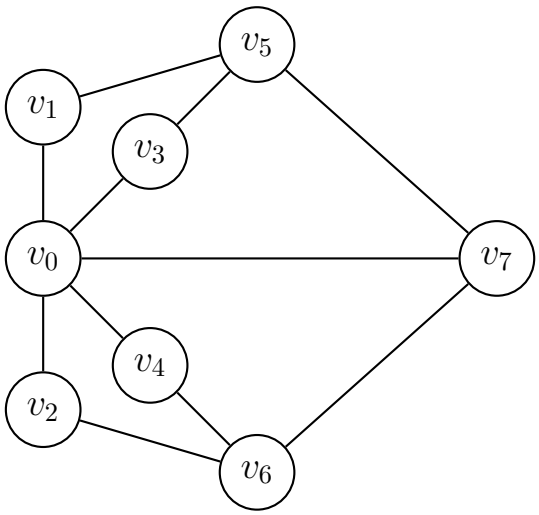
1. Eine mögliche Lösung ist: $(v_1, v_0), (v_2, v_0), (v_0, v_3)$
2. Wir verwenden eine modifizierte Version der Tiefensuche. Diese soll einen Knoten identifizieren, der mindestens zwei eingehende und mindestens

eine ausgehende Kante hat. Zu einem solchen Knoten führt mindestens eine Nichtbaumkante. Also wird immer, wenn eine Nichtbaumkante (u, v) identifiziert wurde, wobei v nicht die Wurzel des Tiefensuchbaumes ist, überprüft, ob v eine ausgehende Kante (v, x) hat. Wenn ja, dann bilden die Kanten $(u, v), (\texttt{parent}[v], v), (v, x)$ ein „Y"-Muster.
Wird eine Nichtbaumkante (u, s) zur Wurzel s des Tiefensuchbaumes gefunden, so wird sich diese Kante gemerkt. Gibt es eine zweite Nichtbaumkante (w, s), so bilden $(u, s), (w, s), (s, x)$, wobei x ein beliebiges Kind von s im Tiefensuchbaum ist, ein „Y"-Muster.
Enthält der Eingabegraph ein „Y"-Muster, so findet dieser Algorithmus es, denn ein Knoten, zu dem zwei Kanten in einem „Y"-Muster zeigen, hat mindestens Eingangsgrad 2 und damit mindestens eine, bzw. zwei im Sonderfall der Wurzel, eingehende Nichtbaumkante.
Da unsere Modifikationen die Laufzeit des Tiefensuche-Algorithmus nicht verändern, liegt die Laufzeit dieses Algorithmus in $\Theta(n + m)$.

Aufgabe 2 - Eulerkreise $(7\ \text{Punkte})$

Gegeben sei ein zusammenhängender Graph $G = (V, E)$. Unter einem Eulerkreis verstehen wir einen Kreis in G, der alle Kanten aus E enthält. Wir wollen uns nun einen Algorithmus überlegen, der in einem gegebenen Graphen einen Eulerkreis bestimmt, sofern er denn existiert.

1. Besitzt der folgende Graph einen Eulerkreis? Falls dem so ist, dann gib die Kantenreihenfolge in diesem Kreis an. Ansonsten begründe kurz, warum dem nicht so ist. (1 Punkt)



2. Zeige: G enthält genau dann einen Eulerkreis, wenn die Kantenmenge E die Vereinigung disjunkter Kreise in G ist. (3 Punkte)
   Hinweis: Disjunkte Pfade sind solche, die sich keine Kanten teilen.

3. Beschreibe einen Algorithmus, der in $\Theta(n+m)$ bestimmt, ob ein gegebener Graph einen Eulerkreis enthält. Begründe kurz Laufzeit und Korrektheit deines Algorithmus. (3 Punkte)

Lösung 2

1. Der Graph hat keinen Eulerkreis. In einem Graphen mit Eulerkreis muss jeder Knoten geraden Grad haben, da wir jeweils immer eine Kante zum Knoten und eine Kante vom Knoten weg folgen, also für jeden Besuch bei dem Knoten zwei Kanten benutzen.
   Dieser Graph hat jedoch 4 Knoten mit ungeradem Grad, nämlich v_0,v_5,v_6 und v_7. Daher kann der Graph keinen Eulerkreis haben.

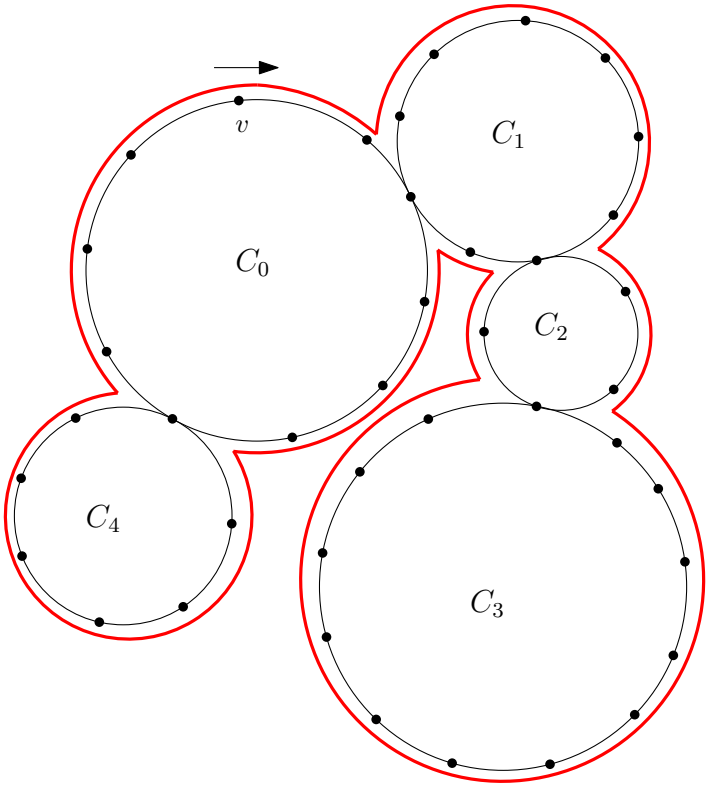
2. $\Rightarrow$:
   Wir machen Induktion über die Anzahl Kanten in dem Graph:

   IA: $n = 0$, das ist trivial, der Graph hat nur einen Eulerkreis, wenn er aus nur einem Knoten v besteht. Der Eulerkreis wäre dann $C = v$ der nur bei einem Knoten bleibt. Dann kann er auch als disjunkte Vereinigung von Kreisen geschrieben werden, denn C ist ein Kreis und auch der einzige also disjunkt zu anderen (nicht existierenden) Kreisen

IV: Die Behauptung hält für alle $k < n$

IS: Wir betrachten den existierenden Eulerkreis $C = v_0, v_1, \cdots, v_n$. Dann muss es mindestens zwei $0 \leq i < j \leq n$ mit $i \neq j$ und $v_i = v_j$ geben. Spätestens $v_0 = v_n$, da ein Eulerkreis wieder zum Start zurückkommt. Unter all diesen Paaren (i, j) betrachte das Paar (i', j') sodass $|i-j|$ minimiert ist. Dann ist $D = v_i, v_{i+1}, \cdots, v_{j-1}, v_j$ ein einfacher Kreis in G und $C' = v_0, v_1, \cdots, v_i, v_{j+1}, \cdots, v_n$ ist ein Eulerkreis ohne den Kreis D. Nun sind C' und D nach Konstruktion kantendisjunkt und teilen sich nur den Knoten v_i. Außerdem hat C' eine Zerlegung in kantendisjunkte Kreise, da C' nun weniger als n Kanten hat und daher die Induktionsvorraussetzung erfüllt. Also ist $C' \cup D$ eine kantendisjunkte Zerlegung in Kreise von C

$\Leftarrow$:
Wir starten bei einem beliebigen Knoten v. Dieser liegt auf einem Kreis $C_0 = (v, x_0, \cdots, x_n, v)$ in G. Der Kreis kann sich Knoten mit anderen Kreisen teilen aber keine Kanten. Sei x_i der erste Knoten im Kreis, der auch auf einem anderen Kreis $C_1 = (x_i, y_0, \cdots, y_n, x_i)$ liegt.
Dieser andere Kreis kann wieder Knoten y_j haben, die in anderen Kreisen C_h enthalten sein können.
Für jeden solchen Kreis, gehen wir die Kanten des Kreises entlang, bis wir einen Knoten finden, der in einem anderen Kreis enthalten ist. Dann gehen wir die Kanten des anderen Kreises entlang.
Sollten wir an einem Knoten ankommen, an dem wir schon einmal waren, so sind wir einen Kreis vollständig entlang gelaufen und können auf den vorigen Kreis zurückkehren. Dieses Verfahren wenden wir an, bis wir wieder bei v angekommen sind.
Da G eine Vereinigung kantendisjunkter Kreise ist, gehen wir somit keine Kante zweimal entlang und da wir bei jedem Treffen auf einen neuen Kreis, diesen auch entlang laufen, entdecken wir damit auch jede Kante.
Unten ist eine Beispieltour für solch einen Graphen gezeichnet.



3. Wir nutzen unser Ergebnis aus Teilaufgabe 2, um zu bestimmen, ob ein gegebener Graph $G = (E, V)$ einen Eulerkreis besitzt. Dazu überprüfen wir, ob E die Vereinigung disjunkter Kreise ist.
   Dazu löschen wir wiederholt Kreise aus G: Solange $|E| > 0$, wählen wir einen beliebigen Knoten v aus G und traversieren von v aus Kanten in G, bis wir wieder bei v sind, also einen Kreis C_v konstruiert haben, der v enthält. Dabei wählen wir jede Kante höchstens ein Mal aus. Wir merken uns die Kanten von C_v und löschen sie anschließend aus G. Wenn es nicht möglich ist, einen Kreis zu v zu konstruieren, bricht der Algorithmus ab um anzuzeigen, dass G keinen Eulerkreis enthält. Ansonsten haben wir E durch die Zuordnung der Kanten in disjunkte Kreise unterteilt und wissen damit, dass G einen Eulerkries hat.
   Dieser Algorithmus liefert das gewünschte Ergebnis: Bricht er nicht ab, so hat er E in disjunkte Kreise zerlegt und damit nach Teilaufgabe 2 einen Eulerkreis gefunden. Angenommen, es existiert ein Eulerkreis zu G, aber der Algorithmus bricht ab. Dann wurde bei der Konstruktion eines Kreises zu einem Knoten v ein Knoten w gefunden, zu dem nur bereits traversierte Kanten inzident sind. Dann muss w ungeraden Grad

haben, was der Annahme widerspräche, dass G einen Eulerkreis hat.
Da wir jede Kante höchstens ein Mal traversieren und anschließend löschen, liegt der Zeitaufwand der Kantenunterteilung und damit auch der Gesamtaufwand unseres Algorithmus in $\Theta(m)$.

Aufgabe 3 - Und täglich grüßt das Murmeltier (25 Punkte)

Die folgenden Probleme lassen sich jeweils mit einem dynamischen Programm lösen. Deine Aufgabe ist es, dir zu überlegen, wie diese Programme aussehen sollten. Beantworte dazu für jedes Problem die folgenden Fragen:

i) Wie lautet die Lösung der Beispielinstanz, die dir gegeben ist?

ii) Wie kann man eine gegebene Probleminstanz in Teilprobleme zerlegen? Wie sieht die Lösung eines Teilproblems aus? Wann kann ein Teilproblem nicht mehr weiter zerlegt werden?

iii) Wie lautet die Rekurrenz, anhand derer die Lösung einer Probleminstanz berechnet werden kann? Benenne dabei explizit die Parameter, die in deiner Rekurrenz vorkommen.

iv) Wie erhalten wir eine tatsächliche Lösung für eine gegebene Probleminstanz? Wie nutze wir dabei die Rekurrenz?

1. MaxSumIncreasingSubarray
   Gegeben sei ein Array $A\colon [\mathbb{Z}; n]$. Wir suchen ein zusammenhängendes Teilarray $A[i \dots j]$ so, dass $A[k] \leq A[k+1]$ für alle $i \leq k < j$ und $\sum_{k=i}^{j} A[k]$ maximal ist.

   Beispielinstanz: $A = \langle -1, 3, 5, -2, 7, -2, 1, 8 \rangle$

2. MinSumSubarray
   Gegeben sei ein Array $A\colon [\mathbb{Z}; n]$. Wir suchen ein zusammenhängendes Teilarray $A[i \dots j]$ so, dass $\sum_{k=i}^{j} A[k]$ minimal wird. Zum Beispiel wäre für $A = \langle 1, 3, 5, -2, 4, -1, -2 \rangle$ das Teilarray $A[5 \dots, 6] = \langle -1, -2 \rangle$ die eindeutige Lösung.

   Beispielinstanz: $A = \langle 1, 8, 4, -1, 3, -2, 1, -2 \rangle$

3. minCostWalk
   Gegeben sei ein Raster mit $n \times m$ Zellen. Jede Zelle (i, j) enthält einen Wert $\texttt{val}(i, j) \in \mathbb{N}$. Wir wollen uns schrittweise durch dieses Raster bewegen, wobei wir uns in einem Schritt immer nur entlang einer Koordinate und nur in positiver Richtung bewegen dürfen. Stehen wir z.B. an Zelle $(3, 4)$, können wir uns nur zu den Zellen $(4, 4)$ oder $(3, 5)$ bewegen. Gesucht ist nun eine Folge von Zellen, wobei aufeinander folgende Zellen jeweils genau einen gültigen Schritt entfernt sind, von der Zelle $(0, 0)$ zur Zelle $(n - 1, m - 1)$ so, dass die Summe der `val`-Werte der durchlaufenen Zellen minimal ist.

   Beispielinstanz: Das folgende Raster mit 3×4 Zellen (die Zelle (0, 0) steht in der linken oberen Ecke):

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

4. LongestCommonSubsequence
   Gegeben seien zwei Arrays $A\colon [\text{char}; n], B\colon [\text{char}; m]$. Wir suchen die längste Zeichenfolge, die sowohl in A als auch in B enthalten ist.
   Beachte: diese Zeichenfolge muss nicht in einem zusammenhängenden Teilarray von A oder B stehen. Die längste gemeinsame Sequenz von $\langle$`s, c, h, a, f, f, n, e, r`$\rangle$ und $\langle$`k, a, r, t, o, f, f, e, l`$\rangle$ etwa ist $\langle$`a, f, f, e`$\rangle$.

   Beispielinstanz:

$$A = \langle \texttt{a, l, p, e, n, g, l, o, c, k, e} \rangle$$

$$B = \langle \texttt{f, r, a, k, t, a, l, g, e, o, m, e, t, r, i, s, c, h} \rangle$$

5. LongestPalindromicSubstring
   Gegeben sei ein Array $A\colon [\text{char}; n]$. Wir suchen das längste zusammenhängende Teilarray von A, welches ein Palindrom bildet.

   Beispielinstanz: $A = \langle$`b, o, s, s, l, e, v, e, l`$\rangle$

Lösung 3

1. MaxSumIncreasingSubarray

   i) $A[6 \dots 7] = \langle 1, 8 \rangle$

   ii) Zu einer gegebenen Instanz, also einem gegebenen Array A, sind die Teilprobleme die Teilarrays von A, die beim Index 0 beginnen. Die Lösung eines Teilproblems $A[0 \dots, i]$ besteht dann aus Indizes, die das aufsteigend sortierte Teilarray von $A[0 \dots i]$ mit der größten Summe der Elemente abgrenzen und w_{max} seine Summe. Außerdem speichern wir uns den Index der längesten aufsteigend sortierten Teilfolge mit der größten Summe, dass mit i endet. Das kleinste Teilproblem ist $A[0 \dots 0] = A[0]$.

   iii) Seien i_{max}, j_{max} die Indizes, die das aufsteigend sortierte Teilarray mit der größten Summe in $A[0 \dots k-1]$ begrenzen und i_{end} der Index, der das aufsteigend sortierte Teilarray mit maximaler Summe, das in $k-1$ endet, begrenzt und w_{end} seine Summe.
   Dann gilt für die neuen Indizes $i'_{max}, j'_{max}, w'_{max}, i'_{end}, w'_{end}$ zum Teilarray $A[0 \dots k]$:

$$(i'_{end}, w'_{end}) = \begin{cases} (i_{end}, w_{end} + A[k]) & \mid A[k] \geq A[k-1] \wedge w_{end} \geq 0 \\ (n, A[n]) & \mid \text{sonst} \end{cases}$$

$$(i'_{max}, j'_{max}, w'_{max}) = \begin{cases} (i'_{end}, n, w'_{end}) & \mid w'_{end} > w_{max} \\ (i_{max}, j_{max}, w_{max}) & \mid \text{ sonst} \end{cases}$$

   iv) Wir starten bei $A[0, 1]$ und berechnen in jedem Schritt die Teillösungen für $A[0 \dots k]$ mithilfe der Rekursion oben. Die gesuchte Lösung ist dann $(i_{max}, j_{max}, w_{max})$ nachdem wir $A[0 \dots n]$ abgearbeitet haben.

2. MinSumSubarray

   i) $A[5 \dots 7] = \langle -2, 1, -2 \rangle$

   ii) Zu einer gegebenen Instanz, also einem gegebenen Array A, sind die Teilprobleme die Teilarrays von A, die beim Index 0 beginnen. Die Lösung eines Teilproblems $A[0 \dots, i]$ besteht dann aus zwei Indizes,

die das Teilarray von $A[0 \dots i]$ mit kleinster Summe der Elemente abgrenzen. Das kleinste Teilproblem ist $A[0 \dots 0] = A[0]$.

iii) Sei $i \in \{0, \dots, n-1\}$. Wir definieren:

$$\texttt{minEndVal}(i) = \begin{cases} A[0] & \mid i = 0 \\ \min\{\texttt{minEndVal}(i-1) + A[i], A[i]\} & \mid \text{ sonst} \end{cases}$$

$$\texttt{minStart}(i) = \begin{cases} 0 & \mid i = 0 \\ \texttt{minStart}(i-1) & \mid \texttt{minEndVal}(i-1) + A[i] < A[i] \\ i & \mid \text{ sonst} \end{cases}$$

Dabei bezeichnet $\texttt{minStart}(i)$ den Index an dem die Teilfolge, die die Lösung für das durch $A[0 \dots i]$ definierte Teilproblem darstellt, beginnt. $\texttt{minEndVal}(i)$ ist die minimale Summe der Elemente einer Teilfolge, die am Index i endet.

iv) Wir können die Teilfolge, die die letztendliche Lösung darstellt, bestimmen, indem wir den Index $i \in \{0, \dots, n-1\}$ finden, der einen minimalen `minEndVal`-Wert hat. Die gesuchte Lösung ist dann $A[\texttt{minStart}(i) \dots i]$.

3. MinCostWalk

i) $\langle (0,0), (0,1), (1,1), (1,2), (1,3), (2,3) \rangle$ (Kosten: 14)

ii) Zu einer gegebenen Instanz, d.h. einem gegebenen Raster, sind die Teilprobleme gegeben durch Teilraster der Größe $i \times j$ mit $i \leq n, j \leq m$, die die Zelle $(0,0)$ enthalten. Die Lösung eines Teilproblems ist dann eine Folge von Zellen von $(0,0)$ zu $(i-1, j-1)$ mit kleinstmöglicher `val`-Summe. Das kleinste Teilproblem ist das Raster, das nur die Zelle $(0,0)$ enthält.

iii) Seien $0 \leq i \leq n, 0 \leq j \leq m$. Wir definieren:

$$\texttt{minWalkLen}(i,j) = \begin{cases} \texttt{val}(0,0) & \mid i = j = 0 \\ \texttt{minWalkLen}(0, j-1) + \texttt{val}(i,j) & \mid i = 0 \\ \texttt{minWalkLen}(i-1, 0) + \texttt{val}(i,j) & \mid j = 0 \\ \min\{\texttt{minWalkLen}(i-1, j) & \\ \quad , \texttt{minWalkLen}(i, j-1)\} & \\ +\texttt{val}(i,j) & \mid \text{ sonst} \end{cases}$$

Dabei bezeichnet $\texttt{minWalkLen}(i,j)$ die minimale Summe von `val`-Werten entlang eines Weges aus gültigen Schritten von $(0,0)$ zu (i,j).

iv) Wir können eine Lösung rekonstruieren, indem wir die Folge von hinten nach vorne anhand der `minWalkLen`-Werte konstruieren. Wir starten bei $(n-1, m-1)$ und bewegen uns immer von der aktuellen Zelle (i,j) zu der Zelle, deren `minWalkLen`-Wert genau $\texttt{minWalkLen}(i,j) - \texttt{val}(i,j)$ ist.

4. LongestCommonSubsequence

i) $\langle$`a, l, g, o`$\rangle$:)

ii) Zu einer gegebenen Instanz, d. h. zu zwei gegebenen Arrays A und B, sind die Teilprobleme gegeben durch zwei Teilarrays $A[0 \ldots, i]$ und $B[0 \ldots j]$ mit $i \leq n, j \leq m$. Die Lösung eines solchen Teilproblems ist dann gegeben durch die längste gemeinsame Teilfolge, die sich $A[0 \ldots, i]$ und $B[0 \ldots j]$ teilen.

iii) Seien $0 \leq i \leq n, 0 \leq j \leq m$. Wir definieren:

$$\texttt{maxSeqLen}(i,j) = \begin{cases} 0 & \mid (i = 0 \vee j = 0) \wedge A[i] \neq B[j] \\ 1 & \mid (i = 0 \vee j = 0) \wedge A[i] = B[j] \\ \texttt{maxSeqLen}(i-1, j-1) + 1 & \mid A[i] = B[j] \\ \max\{\texttt{maxSeqLen}(i-1, j) & \\ \quad , \texttt{maxSeqLen}(i, j-1)\} & \mid \text{ sonst} \end{cases}$$

Dabei bezeichnet $\texttt{maxSeqLen}(i,j)$ die Länge der längsten gemeinsamen Teilsequenz von $A[0 \ldots i]$ und $B[0 \ldots j]$.

iv) Um anhand der `maxSeqLen`-Werte die längste gemeinsame Sequenz von A und B zu bestimmen, sammeln wir sie in A auf. Dazu gehen wir wie folgt vor: Wir starten bei den Indizes $a = n-1, b = m-1$. Gilt $\texttt{maxSeqLen}(a,b) = \texttt{maxSeqLen}(a-1,b)$, so setzen wir nur $a := a-1$, ansonsten setzen wir zusätzlich $b := b-1$ und merken uns davor das Zeichen $A[a]$. Dies setzen wir so lange fort, bis $a = 0$.

5. LongestPalindromicSubstring

i) $\langle$`l, e, v, e, l`$\rangle$

ii) Zu einer gegebenen Instanz, also einem gegebenen Array A sind die Teilprobleme beliebige Teilarrays $A[i \dots j]$ mit $0 \leq i \leq j \leq n-1$. Die Lösung eines Teilproblems $A[i \dots j]$ besteht dann aus Indizes i_{max}, j_{max}, die das längste Palindrom in $A[i \dots j]$ begrenzen.
Ein kleinstes Teilproblem ist ein einzelnes Element in A, also $A[i]$ mit $i \in \{0, \dots, n-1\}$.

iii) Seien $0 \leq i \leq j \leq n-1$. Wir definieren:

$$\texttt{isPalindromic}(i,j) = \begin{cases} \texttt{true} & \mid i = j \\ \texttt{true} & \mid j - i = 1 \wedge A[i] = A[j] \\ \texttt{false} & \mid j - i = 1 \wedge A[i] \neq A[j] \\ \texttt{true} & \mid A[i] = A[j] \wedge \\ & \quad \texttt{isPalindromic}(i+1, j-1) \\ \texttt{false} & \mid \text{ sonst} \end{cases}$$

Dieser Wert gibt an, ob $A[i \dots j]$ ein Palindrom ist.

iv) Wir berechnen $\texttt{isPalindromic}(i,j)$ für alle $0 \leq i \leq j \leq n-1$ anhand der obigen Rekurrenz. Anschließend suchen wir die Indizes i_{max}, j_{max} mit größtmöglicher Differenz, für die der zugehörige `isPalindromic`-Wert `true` ist. Die gesuchte Lösung ist dann $A[i_{max} \dots j_{max}]$.