Woche 9 – Übersicht & Tricks

(Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.)

Anmerkungen zu Serie 6: Allgemeines

- Es gab einige Probleme bei starker Induktion für Invariantenbeweise. Löst dazu noch einmal Aufgaben aus bspw. Altklausuren.
- Bitte achtet darauf, dass eure Lösungen bei DP alle Punkte beantworten. Im Optimalfall sollte eure Lösung exakt das vorgegebene Format erfüllen. In der Klausur bekommt ihr auch das Format als Vorgabe und sollt dann die Lücken füllen – also übt das.
- Bei der Angabe der Berechnungsvorschrift bekommen sehr lange Texte, gespickt mit Beispielen etc., meistens deutlich weniger Punkte als eindeutige mathematische Ausdrücke.

Wiederholung: Dynamic Programming (Palindromic Substrings)

Links: https://leetcode.com/problems/palindromic-substrings/

Die Aufgabenstellung ist sehr simple: Für einen gegebenen nichtleeren String aus lowercase letters, s, soll die Anzahl der Palindrome zurückgegeben werden, die Substrings von s sind. Für jeden nichtleeren String sind das trivialerweise alle Strings der Länge 1.

Für den String aaba wäre das korrekte Resultat bspw.: **6**, denn:

Palindromic Substrings of Length 1: „a“ „a“ „b“ „a“

Palindromic Substrings of Length 2: „aa“

Palindromic Substrings of Length 3: „aba“

Palindromic Substrings of Length 4: -

Als Ansatz verwenden wir diese Rekursion: Substrings der Länge 1 sind palindromic und zählen ins Endresultat mit rein (BASE CASE). Für Substrings s der Länger n > 1 gilt folgendes: s ist palindromic wenn der erste und der letzte Buchstabe im Substring gleich sind und der Substring ohne die beiden Buchstaben palindromic ist. Falls dies der Fall ist, werden sie ins Endresultat reingezählt (REC. CASE).

Also konstruieren wir eine False-Initialisierte 2D Boolean DP-Table der Größe: $n \times n$ (n ist Länge von s) für die gilt: DP[i][j] = True falls der Substring $s[i\ bis\ j]$ palindromic ist und einen Counter „result“. Die Berechnungsvorschrift für einen Entry entnehmen wir direkt der oben beschriebenen Rekursion: Alle Einträge mit $i > j$ ergeben keinen Sinn und bleiben False. Alle Einträge mit $i = j$ sind Base Cases und werden mit True initialisiert. Für alle Einträge mit $i < j$ betrachten wir folgendes: Ist $s[i] == s[j]$? Falls ja, setzen wir DP[i][j] = DP[i+1][j-1] und falls DP[i][j] nun True ist, erhöhen wir den Counter um eins. Ähnlich wie bei MCM fangen wir also mit der Hauptdiagonalen der quadratischen Table an und arbeiten die darüberliegenden Subdiagonalen Stück für Stück ab. Als Code:

(**Lösungscode in JAVA am Ende des Dokuments**)

Graphentheorie –Definitionen:

- **Graph $G = (V, E)$:** Ein Graph besteht aus einer endlichen Menge an Knoten (V) und einer endlichen Menge an Kanten (E).
- **Kanten $e = \{u, v\} \in E$**: Ein unsortiertes Paar aus Knoten $u, v \in V$.
- **Gerichteter Graph $G = (V, E)$:** Besteht aus endlichen Menge an Knoten (V) und einer endlichen Mengen an Kanten (E) wobei jedoch $E \subseteq V \times V$. Die Ordnung der Kanten spielt also eine Rolle: Das heisst übersetzt, dass die Kanten "eine Richtung" haben. Eine Kante (u, v) verläuft von u zu v.
- **Knoten $v \in V$**: Elemente, die unseren Graphen modellieren. Werden normalerweise (falls keine passendere Beschriftung existiert) mit natürlichen Zahlen beschriftet.
- **Inzident:** Eine Kante $e \in E$ ist inzident zu zwei Knoten $u, v \in V$, falls $e = \{u, v\}$.
- **Adjazent:** Zwei Knoten $u, v \in V$ sind adjazent, falls eine Kante $e = \{u, v\} \in E$ existiert.
- **Grad:** Der Grad eines Knoten $v \in V$, $\mathbf{deg}(v)$, ist definiert als die Anzahl an Kanten, die in G zu v inzident sind.
- **Vollständiger Graph:** Ein Graph G heisst vollständig, falls für jedes Paar an Knoten $u, v \in V$ gilt, dass es eine Kante $\{u, v\} \in E$ gibt, also alle Knoten mit allen Knoten verbunden sind. In einem Graphen, der kein Multigraph ist und keine Schleifen enthält, ist dies äquivalent zu der Aussage, dass für alle Knoten $v \in V$ gilt: $\deg(v) = n - 1$ (wenn $|V| = n$).
- **Leerer Graph:** Graph $G = (V, E)$ mit $V, E = \emptyset$.
- **Gewichteter Graph $G = (V, E, c)$:** Ein Graph mit einer Kantengewichtsfunktion $c: E \to R$, die jeder Kante einen Wert zuweist. Besonders interessiert zum Modellieren von Kosten, beispielsweise Wegkosten.
- **Nachbarschaft eines Knoten $N(v)$**: Wir definieren mit $N(v) \coloneqq \{w \in V \mid \{v, w\} \in E\}$, also der Menge aller Knoten, zu denen ein Knoten adjazent ist, als Nachbarschaft dieses Knoten.
- **Nachfolger- / Vorgängermenge $N^+(v), N^-(v)$:** Äquivalent der Nachbarschaft eines Knoten $v \in V$ für gerichtete Graphen, die Richtung der Kanten berücksichtigt.
- **Weg:** Eine Sequenz von Knoten $< v_1, \dots, v_{k+1} >$ für die für jedes $i \in \{1, \dots, k+1\}$ gilt: Es existiert eine Kante $\{v_i, v_{i+1}\} \in E$, das heisst, die Knoten sind über Kanten verbunden.
- **Pfad:** Ein Weg, der keinen Knoten mehrfach verwendet.

- **Zyklus:** Ein Weg $< v_1, \dots, v_{k+1} >$ mit $v_1 = v_{k+1}$, also ein Weg, der dort endet, wo er beginnt. **ACHTUNG:** Auf englisch spricht man hier von einem Circuit.

- **Kreis:** Zyklus, der keine Kante und keinen Knoten mehr als einmal verwendet. Ausnahme ist der Start, bzw. Endknoten, der zweifach genutzt wird. **ACHTUNG:** Auf englisch spricht man hier von einem Cycle.

- **Kreisfreier Graph:** Ein Graph heisst kreisfrei, falls es keinen Knoten gibt, für den ein Kreis existiert.

- **Eulerweg:** Weg, der jede Kante exakt einmal verwendet. Notwendige und Hinreichende Bedingung: Zusammenhängend und alle Knoten bis auf Start- und Endknoten haben geraden Knotengrad. (Beweis: Siehe VL).

- **Eulerzyklus:** Eulerweg bei dem Start- und Endknoten identisch sind. Notwendige und hinreichende Bedingung: Zusammenhängend und alle Knoten haben geraden Knotengrad. (Beweis: Siehe VL).

- **Hamiltonpfad:** Pfad, der jeden Knoten exakt einmal verwendet. (Kein effizienter Algorithmus bekannt).

- **Zusammenhangskomponente:** Menge $X \subseteq V$ an Knoten, sodass für alle $u, v \in X$ gilt: Es existiert ein Pfad zwischen u und v.

- **Zusammenhängender Graph:** Ein Graph heisst zusammenhängend, falls nur eine einzige Zusammenhangskomponente existiert.

- **Erreichbarkeit:** Für zwei Knoten $u, v \in V$ gilt, dass u v erreicht, falls es einen Pfad von u nach v gibt.

Aussagen über Graphen:

Aussage 1: Es existiert ein Eulerkreis gdw. alle Knoten im Graph einen geraden Grad haben.

The Proof is trivial and left as an exer.. just kidding, guckt in die VL-Notizen, dort wird er sehr ausführlich und verständlich beschrieben.

Interessant: Für gerichtete Graphen ist die Bedingung: $\forall v \in V: N^+(v) = N^-(v)$ *was allerdings äquivalent ist, wie beim genauen Hinsehen festgestellt werden kann.*

Aussage 2: Für jeden Graphen gilt: $\sum_{v \in V} \boldsymbol{deg}(\boldsymbol{v}) = \mathbf{2} \cdot |\boldsymbol{E}|$**.**

Beweis: Regel des doppelten Abzählens: Wir zählen für jeden Knoten die inzidenten Kanten, dabei wird jede Kante $\{u, v\} \in E$ *einmal beim Zählen der zu* u *inzidenten Kanten zum Bestimmen von*

$\deg(u)$ *und einmal beim Zählen der zu* v *inzidenten Kanten zum Bestimmen von* $\deg(v)$ *berücksichtigt. So wird jede Kanten zweimal gezählt, also folgt die Aussage. (AnW Skript)*

Aussage 3: Für jeden Graphen gilt: Die Anzahl der Knoten mit ungeradem Grad ist gerade.

Beweis: Sei V_g *die Menge der Knoten mit geradem Grad und* V_u *die Menge der Knoten mit ungeradem Grad. Dann gilt:* $\sum_{v \in V} \deg(v) = \sum_{v \in V_u} \deg(v) + \sum_{v \in V_g} \deg(v) = 2|E|$. *Nun gilt auch: Die Summe von geraden Zahlen ist auch immer eine gerade Zahl, also ist* $\sum_{v \in V_g} \deg(v)$ *gerade. Weiterhin gilt: Die Differenz von geraden Zahlen ist immer gerade, also ist* $2|E| - \sum_{v \in V_g} \deg(v)$ *gerade und daraus folgt die Aussage. (AnW Skript)*

Datenstrukturen für Graphen (Einführung, mehr nächste Woche):

Es gibt zwei wichtige Datenstrukturen, die wir verwenden, um Graphenprobleme algorithmisch zu lösen: Adjazenzmatrizen und Adjazenzlisten. Eine geeignete Datenstruktur sollte Knoten und Kanten – und, falls nötig, Kantengewichte – modellieren und dabei das effiziente Arbeiten mit dem Graphen erlauben.

Adjazenzmatrizen (Einführung):

Eine Matrix A der Grösse $n \times n, n \coloneqq |V|$, bei der gilt: $a_{ij} = 1 \Leftrightarrow \exists(v_i, v_j) \in E$. Also sind Adjazenzmatrizen für ungerichtete Graphen trivialerweise symmetrisch. Eine Adjazenzmatrix benötigt folgenden Platz: $\Theta(|V|^2)$.

Adjazenzlisten (Einführung):

Falls es nur wenige Kanten in unserem Graphen gibt, verschwenden wir mit Adjazenzmatrizen sehr viel Speicherplatz. Die Lösung: Adjazenzlisten: Eine Adjazenzliste ist ein Array, dessen Einträge Listen sind. In $A[i]$ finden wir den Kopf einer verketten Liste, die alle Knoten enthält, zu denen $v_i \in V$ adjazent ist. Falls es sich um einen gerichteten Graphen handelt, dann bloss alle Knoten $u \in N^+(v)$.

Vergleich:

Man kann nicht allgemein sagen, welche Datenstruktur besser ist. Beide haben Vor- und Nachteile. Hier ist eine Auflistung an möglichen Laufzeiten (AnD Skript):

Alle Nachbarn von $v \in V$ ermitteln	$\Theta(n)$	$\Theta(\deg^+(v))$
Finde $v \in V$ ohne Nachbar	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
Ist $(u, v) \in E$?	$\mathcal{O}(1)$	$\mathcal{O}(\deg^+(v))$
Kante einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante löschen	$\mathcal{O}(1)$	$\mathcal{O}(\deg^+(v))$

Achtung: Hier gibt es leider einen Fehler. Es ist eine gute Übung zum Verständnis, den schnell zu finden ;)

Implementation:

Eine Adjazenzmatrix wird üblicherweise ganz normal als 2-dimensionales Array konstruiert. Für eine Adjezenzliste könnte in Java beispielsweise ein Array aus LinkedLists verwendet werden. Per OOP lässt sich das natürlich auch den eigenen Wünschen entsprechend alles selbst sehr gut konstruieren.

Hier eine Beispiel-Implementation: (Quelle: https://www.geeksforgeeks.org/graph-and-its-representations/)

```java
import java.util.*;

class Graph {

    // A utility function to add an edge in an
    // undirected graph
    static void addEdge(ArrayList<ArrayList<Integer> > adj,
                        int u, int v)
    {
        adj.get(u).add(v);
        adj.get(v).add(u);
    }

    // Driver Code
    public static void main(String[] args)
    {
        // Creating a graph with 5 vertices
        int V = 5;
        ArrayList<ArrayList<Integer> > adj
                    = new ArrayList<ArrayList<Integer> >(V);

        for (int i = 0; i < V; i++)
            adj.add(new ArrayList<Integer>());

        // Adding edges one by one
        addEdge(adj, 0, 1);
        addEdge(adj, 0, 4);
        addEdge(adj, 1, 2);
        addEdge(adj, 1, 3);
        addEdge(adj, 1, 4);
        addEdge(adj, 2, 3);
        addEdge(adj, 3, 4);
    }
}
```

Code für Palindromic Substrings:

```java
class Solution {
        public int countSubstrings(String s) {
            int n = s.length();
            int result = 0;
            boolean[][] dp = new boolean[n][n];

            for(int i = 0; i<n; i++){
                dp[i][i] = true;
            }
            for (int dist = 1; dist < n; dist++) {
                for (int i = 0; i+d < n; i++) {
                    int j = i + d;
                    if (s.charAt(i) == s.charAt(j)) {
                        dp[i][j] = dp[i+1][j-1];
                        if (dp[i][j]) result++;
                    }
                }
            }
            return result;
        }
}
```

DP – Programmieraufgabe: Leetcode

In der letzten Übungsstunde haben wir folgende Aufgabe gelöst:

https://leetcode.com/problems/word-break/

Das hier ist ein möglicher Lösungscode:

```java
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {

        //DP[i] = ist es möglich Substring der Länge i mit wordDict zu schreiben.
        boolean[] DP = new boolean[s.length()+1];

        //Base Case
        DP[0] = true;

        for(int i = 0; i<DP.length; i++){
            if(DP[i]){
                for(String word : wordDict){
                    if(i+word.length()<=s.length()
                       && s.substring(i, i+word.length()).equals(word)){
                        DP[i+word.length()] = true;
                    }
                }
            }
        }
        return DP[s.length()];
    }
}
```