Woche 6 – 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 4: Allgemeines

- Korrektheitsbeweise mit Invarianten: Die Invariante (INV(i)) muss stark genug sein, dass die Korrektheit des Algorithmus aus der Gültigkeit der Invariant nach der letzten Iteration folgt.
- Wenn man keine **starke** Induktion verwendet hat, ist es nicht ausreichend, zu sagen "INV(i): A[i] ist he smallest key in A[i,…n]". Denn dann wissen wir nicht, ob das restliche Array sortiert ist oder nicht.
- Pseudocode waren alle ordentlich. Achtet bitte darauf, dass die Algorithmen überhaupt funktionieren können: Pseudocode heisst nicht, einfach mehrere Ideen untereinander hinzuschreiben. Struktur ist wichtig.
- Auf Edge-Cases achten! Immer!
- Programmieraufgaben: "Mein Algo ist doch richtig und trotzdem bekomme ich keine Punkte" - Die Test-Cases die ihr sehen (und mit denen ihr in Expert testen) könnt, sind extrem klein und enthalten normalerweise kaum Edge-Cases. Sie bestätigen euch nur, dass die Idee das Problem "weitestgehend richtig" löst. Allerdings kann euer Algorithmus trotzdem zu langsam sein oder bestimmte Edge-Cases nicht richtig handlen können. Daher: Nur der endgültige Score nach Submission bewertet euren Algorithmus. Also auch an der Prüfung zuerst diesen prüfen, bevor ihr an die nächste Aufgabe geht. Es ist extrem unangenehm, erst kurz vor Schluss der Prüfung die Programmieraufgaben zu submitten und zu realisieren, dass sie keine – oder kaum – Punkte geben. Alles schon vorgekommen.

Dynamic Programming: Allgemeines

Ihr werdet in der Vorlesung 6 verschiedene DP Konzepte kennenlernen (Längste aufsteigende Teilfolge, Längste gemeinsame Teilfolge, Minimale Editierdistanz, Matrixkettenmultiplikation, Subset-Sum, Knapsack). Mit den Ideen hinter diesen Konzepten werdet ihr sehr viele DP-Probleme sehr gut lösen können. Das heisst, wenn ihr ein Problem bekommt, solltet ihr euch zuerst denken (Beispiel):

"Der grösste Turm, den man mit gegebenen Legosteinen, bei denen nur kleinere auf grössere Steine gesetzt werden dürfen, bauen kann? – Kenn ich ein Konzept, dass ein **ähnliches Problem** löst? Klar, das ist ja wie «Längste aufsteigende Teilfolge», nur mit Legosteinen statt Zahlen".

Das funktioniert natürlich nicht immer. Manchmal ist die "Übersetzung" in ein bekanntes Problem nicht so leicht oder mitunter gar nicht wirklich möglich – zum Beispiel weil das Problem eine DP-

Lösung verlangt, die die bekannten Konzepte erweitert oder einfach nichts mit den bekannten Konzepten zu tun hat.

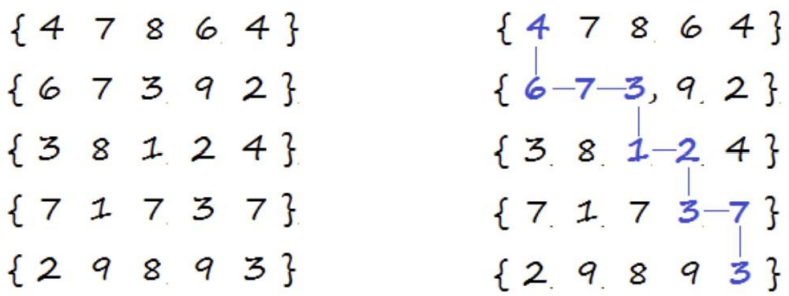
In diesem Fall müsst ihr den Ansatz selbst entwickeln. Das ist aber auch nicht so schwer, wie es am Anfang vorkommt. Ein selbst entwickelter Ansatz besteht immer aus den folgenden Schritten:

1. Überlegen, wie man das Problem in kleinere Teilprobleme zerlegen kann (Divide-and-Conquer Prinzip)
2. Überlegen, wie die Teilprobleme zusammenhängen
3. Überlappende Teilprobleme identifizieren
4. DP-Table konstruieren
5. DP-Table ausfüllen

Hier nun ein Beispiel für ein vergleichsweise einfaches DP-Problem:

Gegeben sei eine Matrix $A \in \mathbb{N}^{m \times n}$, die ein Spielfeld, dass aus $m * n$ Feldern besteht, beschreibt. Jeder einzelne Eintrag der Matrix, $A_{i,j}$ beschreibt die Kosten, um dieses Feld zu traversieren. Wir können von jedem Feld aus horizontal oder vertikal gehen. Alle Kosten sind positiv. Was ist der geringste Preis, um von links oben nach rechts unten zu gelangen?

Beispiel:

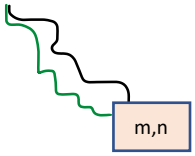


Der Preis des billigsten Weges von $A_{0,0}$ zu $A_{5,5}$ ist hier 36.

Die Lösung:

1. *Überlegen, wie man das Problem in kleinere Teilprobleme zerlegen kann (Divide-and-Conquer Prinzip):*

   Wir sehen: Jede Zelle kann bloss von links oder von oben erreicht werden. (Theoretisch könnte man eine Zelle auch von rechts oder von unten erreichen, aber da wir nach rechts-unten wollen, sind alle Bewegungen nach Oben bzw. nach Links schlecht, da die Kosten immer positiv sind). Also sehen wir für die Kosten, um A_{mn} zu erreichen:
   $Cost(m,n) = A_{m,n} \quad + \quad Min\{Cost(m-1,n), \quad Cost(m,n-1)\}$
   Denn wir wollen ja den geringsten Preis zahlen. In Worten bedeutet das: "Um von $A_{1,1}$ nach $A_{m,n}$ zu gelangen, müssen wir entweder die Kosten des Pfads von $A_{1,1}$ zu $A_{m-1,n}$ plus die Kosten von $A_{m,n}$ bezahlen **oder** die Kosten des Pfads von $A_{1,1}$ zu $A_{m,n-1}$ plus die Kosten von $A_{m,n}$ bezahlen. Und da wir uns für den billigsten Pfad interessieren, nehmen wir von den beiden Möglichkeiten die billigere."



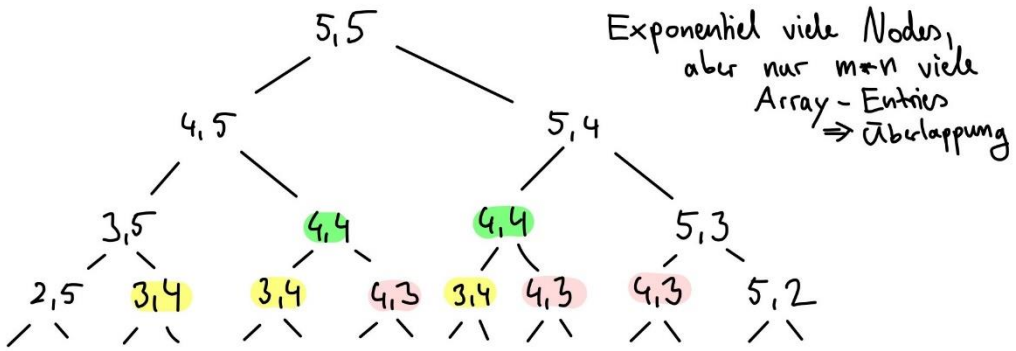
Hier bildlich: Der Preis zu $A_{m,n}$ ist der Preis des grünen Pfades plus der Preis des schwarzen Pfades. Natürlich gibt es noch viel mehr Möglichkeiten für Pfade, aber direkt zu $A_{m,n}$ kommt man nur über 2 Nachbarfelder.

2. *Überlegen, wie die Teilprobleme zusammenhängen:*

   Das haben wir in diesem Fall schon im ersten Schritt weitestgehend festgestellt. Der billigste Pfad zu einem Feld kann in immer als optimale Wahl von den beiden billigsten Pfäden zu seinen beiden Nachbarn betrachtet werden. Der Preis eines Feldes ist dann der Preis des billigeren Pfad zu einem seiner beiden Nachbarn + der Preis von sich selbst.

3. *Überlappende Teilprobleme identifizieren:*

   Wir können uns als Beispiel einen Berechnungsbaum skizzieren:



   Wir sehen: Da es in dem Berechnungsbaum exponentiell viele Nodes gibt, aber unser Array nur $m * n$ viele Felder hat, muss es sein, dass wir den günstigsten Pfad zu bestimmten Feldern mehrfach berechnen. Das sind überlappende Teilprobleme.

4. *DP-Table konstruieren:*

   Wir orientieren uns ganz einfach an unserer Berechnungsvorschrift: Wir erstellen eine $m * n$ grosse Matrix als DP-Table. Ein Eintrag `DP[i][j]` dieser Matrix speichert Preis des billigsten Pfades von $A_{1,1}$ zum Feld $A_{i,j}$.

5. *DP-Table ausfüllen:*

   Wie ihr (Verweis Zusammenfassung Woche 5) bereits wisst, gibt es zwei Wege, DP-Probleme zu lösen: **Memoization** und **Bottom-Up**. Betrachten wir einfach beide Möglichkeiten:

   - Memoization:
     Ihr schreibt einen rekursiven (iterativ geht es natürlich auch) Algorithmus, $cost(m, n)$ berechnet und dabei jeden neu berechnen Wert in der DP-Matrix speichert – und jedes Mal, bevor er eine neue Lösung berechnet, prüft, ob es diesen Wert nicht schon in der DP-Matrix gibt, er also schon berechnet wurde.

   - Bottom-Up:
     Anstatt mit der Berechnung von $cost(m, n)$ zu beginnen, berechnen wir einfach erst alle anderen Pfade – da wir ja wissen, dass wir diese später eh gebrauchen werden. Dafür müssen wir allerdings zunächst ein paar Base-Cases definieren, auf denen wir aufbauen können: $cost(1,1) = A_{1,1}$. Jetzt können wir durch die DP-Table iterieren und für jeden Entry mithilfe unserer Berechnungsvorschrift

die Kosten bestimmen. Dabei ist es hier offensichtlich am sinnvollsten, Zeile für Zeile zu iterieren, da wir für die berechnung eines Feldes, seinen linken und seinen oberen Nachbarn brauchen. Die finale Lösung finden wir dann in `DP[m][n]`.

ACHTUNG: ICH HABE HIER IMMER EIN 1-INDEXING VERWENDET

Dynamic Programming: Lösungsformat in Aufgaben und Klausur:

In Aufgaben oder an der Prüfung wird von euch erwartet, dass ihr kurz und deutlich mit diesen Punkten euren DP-Algorithmus erklärt:

1. Dimensions of DP-Table
2. Definition of the DP-Table (What is the meaning of an entry)
3. Computation of an entry ("Berechnungsvorschrift")
4. Calculation order ("Which order allows the use of computations in previous steps")
5. Extracting the Solution ("Where in the DP-Table can you find the solution")
6. Runtime

Longest Increasing Subsequence (LIS/LAT):

Wir verwenden eine $1 \times n$ – DP-Table. In dem i-ten entry speichern wir das letzte Element einer aufsteigenden Teilfolge der Länge i. Diese DP-Table ist offensichtlich aufsteigend sortiert. Zur Berechnung des Arrays gibt es zwei Möglichkeiten, eine ist besser: Wir iterieren über das gegebene Array und gucken uns in der k-ten Iteration das Element a_k an. Dann suchen wir mit binärer Suche in unserer DP-Table die längste Teilfolge, an die a_k noch angehängt werden kann. Dann ersetzen wir $DP[m+1]$ durch das Element a_k **falls** $a_k < DP[m+1]$. Am Anfang setzen wir DP[0] auf -Inf und DP[k] = Inf für alle $k \leq n$. Die Lösung können wir schlussendlich finden, indem wir durch die DP-Table iterieren und den Index der längsten Teilfolge, für die wir ein Element haben, zurückgeben. Das ganze benötigt in n Iterationen je eine Binäre Suche, läuft also in $O(n * \log n)$.

Longest Common Subsequence (LCS/LGT):

Gegeben sind Wörter/Sequencen/Arrays A, B. Wir verwenden eine $m \times n$ – DP-Table. Das (i,j)-te Entry speichert die LCS für die Teilwörter $A[:i], B[:,j]$. Die Berechnungsvorschrift ist offensichtlich: Die LCS von zwei Wörtern $A[:i], B[:,j]$ ist $\max\{1 + LCS(A[:i-1], B[:,j-1])$ (falls $a_i = b_j$), $LCS(A[:i], B[:,j-1]),\ LCS(A[:i-1], B[:,j])\}$. Denn für zwei Wörter der Längen i,j ist die Teilfolge entweder die längste Teilfolge in den beiden möglichen Wortverkürzungen – oder die Folge, die an (i,j) endet. Da wir wieder links/oben/diagonal-oben stehende Einträge für das Berechnen des Entry $DP[i][j]$ brauchen, ist zeilenweises iterieren hier die beste Wahl. Die Lösung kann letzten Endes aus $DP[m][n]$ ausgelesen werden. Da wir über das gesamte Array iterieren, ist dieser Algorithmus in $O(m * n)$

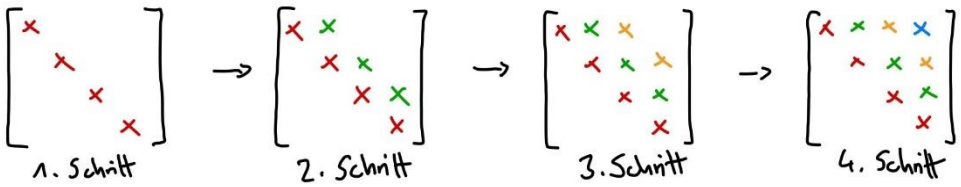
Minimum Edit Distance (MED):

Gegeben sind sind Wörter/Sequencen/Arrays A, B. Wir verwenden eine $m \times n$ – DP-Table. Das (i,j)-te Entry speichert die MED für die Teilwörter $A[:i], B[:,j]$. Die Berechnungsvorschrift ist offensichtlich: Die MED von zwei Wörtern $A[:i], B[:,j]$ ist $\min\{MED(A[:i-1], B[:,j-1])$ (falls $a_i = b_j$), $MED(A[:i], B[:,j-1]) + 1,\ MED(A[:i-1], B[:,j]) + 1\}$. Denn für zwei Wörter der Längen i,j ist die MED entweder die MED der beiden Wortkürzungen plus das Einfügen je eines weiteren Buchstabens/Zeichens oder die MED für beide Wortkürzungen ohne editing, falls die Buchstaben/Zeichen gleich sind. Da wir wieder links/oben/diagonal-oben stehende Einträge für das Berechnen des Entry $DP[i][j]$ brauchen, ist zeilenweises iterieren hier die beste Wahl. Die Lösung

kann letzten Endes aus $DP[m][n]$ ausgelesen werden. Da wir über das gesamte Array iterieren, ist dieser Algorithmus in $O(m * n)$

Matrix Chain Multiplication (Konzept):

Gegeben ist eine Sequenz aus Objekten, $A_1, \dots, A_n$, die miteinander verrechnet werden, wobei die Reihenfolge der Berechnung entscheidend ist, beispielsweise Matrizen – oder Faktoren. Wir verwenden eine $n \times n$ – DP-Table. Das (i, j)-te Entry speichert die Kosten / optimale Lösung der Sub-berechnung von A_i bis A_j. Wir berechnen es folgendermassen: wir können die Sequenz von i nach j immer in je zwei Teil-Sequenzen aufteilen. Die optimale Lösung ist dann die Kosten/optimale Lösung der Teilsequenzen + die Kosten/optimale Lösung der Verrechnung der beiden Teilsequenzen. Das berechnen wir für jeden Trenner und dann nehmen wir davon den optimalen (normalerweise den maximalen) Wert. Also formal: $DP[i][j] = \min_{i \le p < j} \{M[i][p] + M[p+1][j] + Verrechnungskosten\}$. Wir berechnen also nur die rechts-obere Dreiecksmatrix in DP. Dafür berechnen wir erst die Hauptdiagonale (Base Cases) und arbeiten und uns dann anschliessend in jeder "grossen" Iteration je einen Schritt in der Matrix nach rechts.



Extra Material:

Klausur FS18: (Programmieraufgabe T1)

ALGO-Turm

Gegeben sei eine Kiste mit $n \le 5000$ *ALGO-Steinen*, welche von 1 bis n durchnummeriert sind (siehe untenstehendes Beispiel). Der i-te ALGO-Stein ist ein Quader mit Grundfläche $\ell_i \times b_i$ und Höhe h_i, mit $\ell_i, b_i \in \{1, 2, \dots, 10000\}$ und $h_i \in \{0, 1, ..., 1000\}$.

Aus diesen Steinen soll nun ein *möglichst hoher* Turm gebaut werden. Dazu dürfen die vorhandenen Steine aufeinandergestapelt werden, solange dabei *keine Überhänge* entstehen. Formaler: Der i-te Stein darf genau dann auf den j-ten Stein aufgesetzt werden, wenn mindestens eine der folgenden zwei Bedingungen gilt:

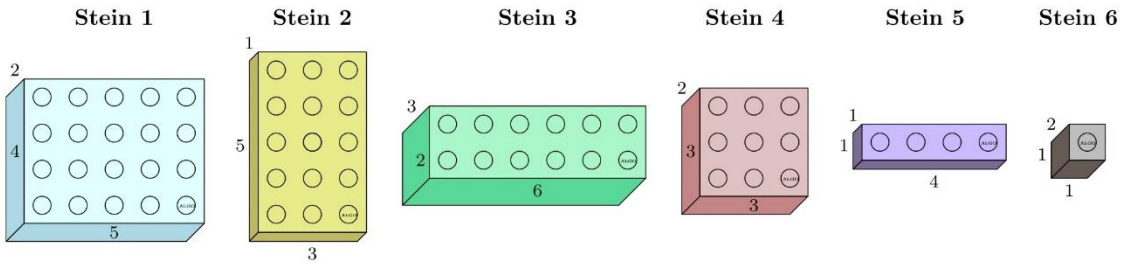
- $\ell_j \ge \ell_i$ und $b_j \ge b_i$; oder
- $\ell_j \ge b_i$ und $b_j \ge \ell_i$.

Intuitiv gesprochen entspricht die zweite Bedingung einer Drehung des i-ten Steins um 90° Grad.

Ein *ALGO-Turm* ist eine Liste $\langle t_1, t_2, ..., t_k \rangle$ von paarweise unterschiedlichen Indizes, sodass für alle $1 \le i < k$ der Stein t_{i+1} auf den Stein t_i aufgesetzt werden darf. Die Höhe H eines solchen Turms ergibt sich als Summe der Höhen aller verwendeten Steine, das heisst $H := \sum_{i=1}^{k} h_{t_i}$.

Ihre Aufgabe ist es nun, die Höhe eines höchstmöglichen ALGO-Turms zu berechnen, welcher mit den vorhandenen ALGO-Steinen gebaut werden kann. Dabei dürfen Sie annehmen, dass alle Grundflächen paarweise unterschiedlich sind, und dass die Steine nach absteigender Grundfläche sortiert sind. In anderen Worten gilt für $1 \le i < n$, dass $\ell_i \cdot b_i > \ell_{i+1} \cdot b_{i+1}$.

Beispiel



Die oben abgebildeten $n = 6$ ALGO-Steine haben die folgenden Dimensionen:

Stein 1 $\ell_1 = 5$, $b_1 = 4$, $h_1 = 2$;

Stein 2 $\ell_2 = 3$, $b_2 = 5$, $h_2 = 1$;

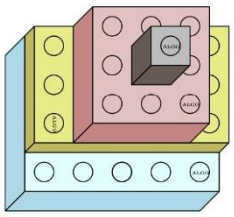
Stein 3 $\ell_3 = 6$, $b_3 = 2$, $h_3 = 3$;

Stein 4 $\ell_4 = 3$, $b_4 = 3$, $h_4 = 2$;

Stein 5 $\ell_5 = 4$, $b_5 = 1$, $h_5 = 1$;

Stein 6 $\ell_6 = 1$, $b_6 = 1$, $h_6 = 2$;

Der höchste ALGO-Turm, welcher mit diesen Steinen gebaut werden kann, ist demzufolge $\langle 1, 2, 4, 6 \rangle$, siehe auch die untenstehende Abbildung. Er hat Höhe $h_1 + h_2 + h_4 + h_6 = 2 + 1 + 2 + 2 = 7$.



Hierbei musste Stein 2 (in gelb) *gedreht* werden, um ihn auf Stein 1 (in blau) aufsetzen zu können.

```java
import java.io.*;
import java.util.LinkedList;
import java.util.Scanner;
import java.lang.Math;
import java.lang.Integer;
import java.lang.String;

class Main {
    public static void main(String[] args) {
        // Uncomment the following two lines if you want to read from a file
        In.open("public/custom.in");
        Out.compareTo("public/custom.out");

        // Read the number of test cases
        int ntestcases = In.readInt();
        for(int t=0; t<ntestcases; t++)
        {
          int n = In.readInt();

          int[] l = new int[n+1];
          int[] b = new int[n+1];
          int[] h = new int[n+1];

          for(int i=1; i<=n; i++)
            l[i] = In.readInt();

          for(int i=1; i<=n; i++)
            b[i] = In.readInt();

          for(int i=1; i<=n; i++)
            h[i] = In.readInt();

          Out.println(solve(n, l, b, h));
        }

        // Uncomment the following line if you want to read from a file
        // In.close();
    }



    //n bezeichnet die Anzahl der ALGO Steine.
    //l[i] enthält die Länge des i-ten ALGO Steins, für i=1,...,n
    //b[i] enthält die Breite des i-ten ALGO Steins, für i=1,...,n
    //h[i] enthält die Höhe des i-ten ALGO Steins, für i=1,...,n
    static int solve(int n, int[] l, int[] b, int[] h)
    {
          //TODO: Geben Sie die Höhe des höchsten ALGO Turms, der aus den vorhandenen Steinen
          //gebaut werden kann, zurück.
          int[] LIS = new int[n+1];
          for(int i = 0; i<LIS.length; i++){
            LIS[i] = h[i];
          }
          int iP = 2, jP = 1, max = 0;
          while(iP<n+1){
            jP = 1;
            while(jP<iP){
              if(fits(jP, iP, b, l, h)){
                LIS[iP] = Math.max(LIS[iP], LIS[jP] + h[iP]);
              }
              jP++;
            }
            iP++;
          }
          for(int i = 0; i<LIS.length; i++){
            max = Math.max(max, LIS[i]);
          }
          return max;
    }
    static boolean fits(int j, int i, int[]b, int[]l, int[]h){
      return((l[j] >= l[i] && b[j] >= b[i]) || (l[j] >= b[i] && b[j] >= l[i]));
    }


}
```

Quellen:

Skript (Wie immer)

https://www.techiedelight.com/find-minimum-cost-reach-last-cell-matrix-first-cell/