Woche 7 – 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.)

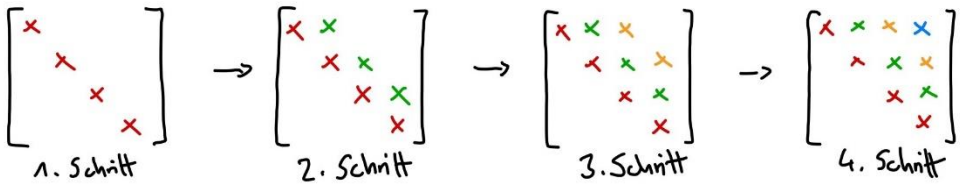
Anmerkungen zu Serie 5: Allgemeines

- Die letzte Serie wurde allgemein sehr gut gelöst, ich bin zufrieden und habe keine allgemeinen Punkte zu bemängeln 😊
- OK, doch: Schreibt bitte sauber, es ist extrem mühsam geschmierten Text zu verstehen.

Matrix Chain Multiplication:

Gegeben ist eine Sequenz aus Objekten, $A_1, \dots, A_n$, die miteinander verrechnet werden, wobei die Reihenfolge der Berechnung entscheidend ist, beispielsweise Matrizen (verschiedene Matrix-Vektor-Multiplikationen kosten unterschiedlich viele Operationen) – oder Faktoren und Summanden (Wenn wir einen Ausdruck maximieren oder minimieren wollen – siehe *Beispiel 2*). Wir verwenden eine $n \times n$ – DP-Table. Das (i, j)-te Entry speichert die Kosten / optimale Lösung der Sub-berechnung der Sequenz 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 jedes Teil und dann nehmen wir davon den optimalen (normalerweise den minimalen) 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 der DP-Table. 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.



Beispiel 2, Dynamic Programming:

Gegeben sind zwei Arrays: Ein n grosses Operanden-Array ($\mathbb{Z}$) und ein $n-1$ grosses Operatoren-Array (Nur $+$ oder $*$) . Also beispielsweise:

Operanden: [7, 4, -3, 6, -5] und Operatoren: [+, +, *, +]

Diese zwei Arrays repräsentieren einen Ausdruck, hier beispielsweise:

$$7 + 4 + (-3) * 6 + (-5)$$

Nun stellt sich folgende Frage: Wie können wir Klammern setzen, um diesen Ausdruck zu maximieren?

Für unser gegebenes Beispiel ist eine optimale “Klammerung”:

$$\Big(\big((7+4)+(-3)\big)*6\Big)+(-5)=43$$

Indem wir alle möglichen Klammern ausprobieren, können wir eine Optimale Lösung finden. Das würde allerdings einen exponentiellen Rechenaufwand bedeuten. Es gibt eine bessere Lösung, die auf der Idee der Matrix Chain Multiplication beruht: Bei allen gegebenen Operatoren handelt es sich um binäre Operatoren, die zwei Operanden als Input nehmen und einen Output geben.

Also gilt für die endgültig maximale Lösung, dass es einen “linken” und einen “rechten” Term gibt, die durch einen Operator miteinander verknüpft werden, also:

$$(\text{linker Term})\ <Operator>\ (\text{rechter Term})$$

In unserem der Lösung zu unserem obigen Beispiel ist dies:

$$\Big(\big((7+4)+(-3)\big)*6\Big)+(-5)$$

Wobei ich hier nun den linken und den rechten Term farblich deutlich gemacht hab und den Operator in schwarz gelassen habe.

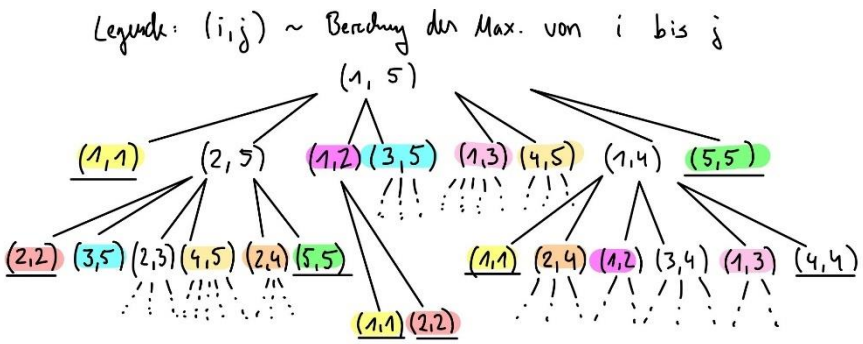
Das heisst, wir müssen für die endgültige Lösung herausfinden, was genau der Operator ist, der unsere Expression in einen Linken und Rechten Term aufteilt – bzw. was genau der linke und der rechte Term sind, deren Verrechnung mit dem Operator den endgültigen Wert maximiert.

Und genau diese Idee “des letzten Schrittes” nutzen wir nun für die Rekursion:
Das ist im Übrigen immer der beste Weg, um eine DP-Rekursion zu entwickeln

$$Max(Term(1\text{ bis }n))=\max_{1\le k\le n}\Big[\max\big(Term(1\text{ bis }k)\big)<Operator\ k>\max\big(Term(k+1\text{ bis }n)\big)\Big]$$

(Wenn du schon etwas weiter gedacht hast, wird dir aufgefallen sein, dass diese Rekurrenzgleichung in diesem Fall das Problem nicht löst – da wir die negativen Werte speziell handlen müssen).

Stell dir vor, wir hätten nun diese Rekursionsgleichung als Algorithmus formuliert (nach dem gleichen Rezept, wie wir es auch für den Fibonacci etc. gemacht haben). Dann könnte ein Berechnungsbaum für obiges Beispiel so aussehen:



Wir sehen: es gibt wieder extrem viele überlappende Subprobleme (farblich markiert). Und das ist nicht einmal der ganze Baum! Also speichern wir doch einfach die Teilergebnisse ab, damit wir sie nicht doppelt berechnen müssen!

Dafür entwerfen wir folgende DP-Table: MAX[][]. Wir speichern ab: MAX[i][j] = Maximum des Terms von i bis j.

Es gibt noch ein Problem, um das wir uns gleich kümmern werden, aber grundsätzlich haben wir damit eine gute DP-Lösung gefunden! Es steht uns natürlich wieder frei, die Table bottom up (nach dem Prinzip der Matrix Chain Multiplication) zu füllen, also erst die Hauptdiagonale MAX[i][i], dann MAX[i][i+1], etc. etc. etc. zu berechnen – oder einfach unseren rekursiven Algorithmus mit Memoization zu verwenden – und jedes Mal, wenn wir das Maximum eines Sub-Terms brauchen, erst zu schauen, ob wir das nicht schonmal berechnet haben, also, ob wir in MAX[i][j] schon einen Eintrag haben.

Sind wir fertig?

Nein, leider nicht: Das Problem, das jetzt noch verbleibt, ist der Grund, weswegen diese Aufgabe als "schwierig" gilt. Aber, gute Nachrichten: Die Ursache des Problems hat nix mehr mit DP – sondern nur mit Schul-Mathematik zu tun:

Wenn wir Zahlen aus zwei Intervallen $[a, b]$, $[c, d]$ miteinander multiplizieren, ist das Maximum $b * d$, falls a, b, c, d alle positiv sind. Falls a, b, c, d jedoch alle negativ sind, ist das Maximum $a * c$ – also das Produkt der beiden **minimalen Teilterme**. Und in allen anderen Fällen ist es $b * c$ oder $a * d$.

Also reicht es nicht, nur die maximalen Teilterme zu speichern. Wir müssen auch noch alle minimalen Teilterme speichern. Und jedes Mal, wenn wir das Maximum eines Terms berechnen, müssen wir gucken, ob es nicht besser ist, die beiden minimalen Teilterme miteinander zu verrechnen. Das ganze resultiert dann in folgender Rekursionsgleichung:

$$Max\left(Term_{\{1\ bis\ n\}}\right)$$
$$= \max_{1 \le k \le n} \begin{cases} \max \begin{bmatrix} \max\left[\max\left(Term_{\{1 \text{ bis k}\}}\right) * \max\left(Term_{\{k+1 \text{ bis n}\}}\right), \min\left(Term_{\{1 \text{ bis k}\}}\right) * \min\left(Term_{\{1 \text{ bis k}\}}\right)\right], \\ \max\left[\max\left(Term_{\{1 \text{ bis k}\}}\right) * \min\left(Term_{\{k+1 \text{ bis n}\}}\right), \min\left(Term_{\{1 \text{ bis k}\}}\right) * \max\left(Term_{\{1 \text{ bis k}\}}\right)\right] \end{bmatrix} \\ if\ Operator\ " * " \\ \max\left(Term_{\{1 \text{ bis k}\}}\right) + \max\left(Term_{\{k+1 \text{ bis n}\}}\right) \\ if\ Operator\ " + " \end{cases}$$

$$Min\left(Term_{\{1\ bis\ n\}}\right)$$
$$= \min_{1 \le k \le n} \begin{cases} \min \begin{bmatrix} \min\left[\max\left(Term_{\{1 \text{ bis k}\}}\right) * \max\left(Term_{\{k+1 \text{ bis n}\}}\right), \min\left(Term_{\{1 \text{ bis k}\}}\right) * \min\left(Term_{\{1 \text{ bis k}\}}\right)\right], \\ \min\left[\max\left(Term_{\{1 \text{ bis k}\}}\right) * \min\left(Term_{\{k+1 \text{ bis n}\}}\right), \min\left(Term_{\{1 \text{ bis k}\}}\right) * \max\left(Term_{\{1 \text{ bis k}\}}\right)\right] \end{bmatrix} \\ if\ Operator\ " * " \\ \min\left(Term_{\{1 \text{ bis k}\}}\right) + \min\left(Term_{\{k+1 \text{ bis n}\}}\right) \\ if\ Operator\ " + " \end{cases}$$

Die sieht sehr kompliziert aus, berücksichtigt beim Berechnen von Max/Min eines Terms aber nur die Schul-Mathematik Regeln, die ich zuvor beschrieben habe.

Wir können diese Rekursionsgleichung nach dem Gleichen Prinzip wie zuvor mit einem rekursiven Algorithmus implementieren – aber anstatt einer DP-Table verwenden wir einfach 2: Eine für die Maxima und eine für die Minima.

Hier eine vollständige Implementation des DP-Algorithmus. Beim genauen Hinsehen kann die Rekursionsgleichung leicht wiederentdeckt werden:

```
private static int maximizing_an_expression(int n, int[] Operand, char[] Operator)
    {
      // "Operand" stores n Operands
      // "Operator" stores n-1 Operators
      // For the expression "1+3*2", Operand={1,3,2} and Operator={+,*}


      int[][] MinTB=new int[n][n];
      int[][] MaxTB=new int[n][n];

      //Initialization (Base Cases):
      for(int i = 0; i<n; i++){
        MaxTB[i][i] = Operand[i];
        MinTB[i][i] = Operand[i];
      }
      int currMax = 0;
      int currMin = 0;
      //DP:
      for(int diff = 1; diff<n; diff++){ //So gehen wir üblich bei MatrixChainMult. vor: Wir loopen über die Diff..
        for(int i = 0; i<n; i++){
          //Fitting intervals:
          if(i+diff<n){

            //Hier initialisiere ich nur currMax und currMin
            if(Operator[i] == '*'){
                currMax = max(MinTB[i][i], MinTB[i+1][i+diff], MaxTB[i][i], MaxTB[i+1][i+diff]);
                currMin = min(MinTB[i][i], MinTB[i+1][i+diff], MaxTB[i][i], MaxTB[i+1][i+diff]);
              }
              else{
                currMax = MaxTB[i][i] + MaxTB[i+1][i+diff];
                currMin = MinTB[i][i] + MinTB[i+1][i+diff];
              }


            for(int k = i; k<(i+diff); k++){
              if(Operator[k] == '*'){
                currMax = Math.max(currMax, max(MinTB[i][k], MinTB[k+1][i+diff], MaxTB[i][k], MaxTB[k+1][i+diff]));
                currMin = Math.min(currMin, min(MinTB[i][k], MinTB[k+1][i+diff], MaxTB[i][k], MaxTB[k+1][i+diff]));
              }
              else{
                currMax = Math.max(currMax, MaxTB[i][k] + MaxTB[k+1][i+diff]);
                currMin = Math.min(currMin, MinTB[i][k] + MinTB[k+1][i+diff]);
              }
            }
            MaxTB[i][i+diff] = currMax;
            MinTB[i][i+diff] = currMin;
          }
        }
      }
      // Please complete this Method.
      return MaxTB[0][n-1];
    }

    private static int max(int min1, int min2, int max1, int max2){
      int curr1 = Math.max(min1*min2, max1*max2);
      int curr2 = Math.max(min1*max2, max1*min2);
      return Math.max(curr1, curr2);
    }

    private static int min(int min1, int min2, int max1, int max2){
      int curr1 = Math.min(min1*min2, max1*max2);
      int curr2 = Math.min(min1*max2, max1*min2);
      return Math.min(curr1, curr2);
    }
```

Subset-Sum:

Ich werde im Folgenden das Problem vorstellen und zeigen, wie man es in Teilprobleme zerlegen kann. Technische Details lasse ich aus, da sie im Skript gefunden werden können:

Gegeben ist ein Array A aus Zahlen und eine Summe S. Wir fragen uns: Gibt es eine Auswahl an Werten in A, sodass die Summe dieser Werte gleich S ist? Dazu könnten wir theoretisch alle mögliche Auswahlmöglichkeiten betrachten, das würde jedoch exponentiellen Rechenaufwand verlangen. Also gehen wir anders vor und überlegen, wie wir das Problem in Teilprobleme zerlegen können – und wie wir Überlappung der Teilprobleme vermeiden können:

Wir sehen folgende Rekursion: Wenn wir das i-te Element betrachten, kann es entweder Teil der Auswahl sein – oder eben nicht:

$$isSum(A,S,i) = \begin{cases} isSum(A,S-A[i],i-1) \ \vee \ isSum(A,S,i-1), & if\ S>0 \\ True, & if\ S==0 \\ False, & if\ S>0 \text{ and } i==0 \\ False, & if\ S<0 \end{cases}$$

Das Problem sind wieder einmal die überlappenden Teilprobleme. Also speichern wir alle Zwischen-Ergebnisse in einem zweidimensionalen Array, unserer DP-Table ab und es gilt offensichtlich: DP[i][j] ist true, falls es eine Auswahl in A[1,…,i] gibt, sodass diese Auswahl gleich Summe j ist. Jetzt können wir die Rekursion mit dieser Table per Memoization effizient implementieren – oder einfach bottom up mit folgender Vorschrift berechnen:

$$DP[i][j] = \begin{cases} DP[i-1][j], & falls\ A[i]>j \\ DP[i-1][j] \ || \ DP[i-1][j-A[i]], & falls\ A[i]\leq j \end{cases}$$

Genaueres zur Initialisierung kann im Skript gefunden werden. Bei der Implementierung auf Indexing achten!

Knapsack:

Gegeben sind zwei Arrays, $values$ und $weights$, die den Ort unseres Verbrechens modellieren: In $values[i]$ speichern wir den der Wert des i-ten Gegenstandes ab und in $weights[i]$ das Gewicht desselben. Des Weiteren ist ein Wert, W, gegeben, der unsere Tragfähigkeit limitiert. Wir wollen maximalen Profit machen. Wie gehen wir vor?

Ähnlich wie beim Subset-Sum Problem könnten wir uns jetzt hinsetzen und alle Möglichkeiten betrachten, wie wir Elemente in unseren Rucksack stecken, ohne das Maximalgewicht zu überschreiten. Allerdings würde es auch hier exponentiellen Rechenaufwand erfordern, also müssen wir effizienter vorgehen und überlegen uns folgende Rekursion:

$$knapsack(A,W,n) = \begin{cases} \max\big(knapsack(A,W-weights[n],n-1)+values[n],\ knapsack(A,W,n-1)\big), & if\ W>0 \\ 0, & if\ W\leq 0 \end{cases}$$

Diese Rekursion können wir nun wieder als Grundlage für eine DP-Table verwenden, um das Berechnen von überlappenden Teilproblemen zu vermeiden:

Wir konstruieren eine DP Table der Dimensionen $n \times W$ und speichern in $DP[i][j]$ den maximalen Profit für die Gegenstände $1, \dots, i$ bei limitierenden Gewicht j ab.

Als Berechnungsvorschrift ergibt sich dann folgendes:

$$DP[i][j] = \max(DP[i-1][j],\ DP[i-1]\big[j-weights[i]\big]+values[i])$$

Falls $j - weights[i] \geq 0$, sonst setzen wir: $DP[i][j] = DP[i-1][j]$.

Beispiel 3 (House Robber):

(Aus dem Live-Programmieren in der Übungsstunde): Die Idee ist einfach:

$$maxDieb(i) = \max\left(values[i] + DP[i-2], DP[i-1]\right)$$

Hier der Code:

```java
class Solution {
    public int rob(int[] nums) {
        int[] DP = new int[nums.length];
        if(nums.length > 0)
            DP[0] = nums[0];
        else
            return 0;
        if(nums.length > 1)
            DP[1] = Math.max(nums[1], nums[0]);
        for(int i = 2; i<nums.length; i++){
                DP[i] = Math.max(DP[i-2] + nums[i], DP[i-1]);
        }
        int ret = 0;
        for(int i = 0; i<nums.length; i++){
            ret = Math.max(ret, DP[i]);
        }
        return ret;
    }
}
```

Beispiel 4 (Bonusaufgabe HS21):

The coffeeshop chain Starduck's is planning to open several cafés in Bahnhofstrasse Zürich. There are n possible locations $1, \ldots, n$ for their shops on Bahnhofstrasse, ordered by their distance to Zürich main station $m_1 < \ldots < m_n$. Opening a shop at location i would yield Starduck's a profit of $p_i > 0$. However, they are not allowed to open cafés that are too close to each other, namely any two cafés should have distance at least d from each other, for some given value $d > 0$.

Provide an algorithm using dynamic programming that computes the maximum total profit that Starduck's can make on Bahnhofstrasse. In order to get full points, your algorithm should have $O(n \log n)$ runtime.

Dimension der DP-Table: Sie ist linear und enthält Einträge von $DP[0]$ bis $DP[n]$, also ist die Grösse $n + 1$.

Definition der DP-Table: $DP[i]$ enthält den maximalen Profit, der mit den Locations $1, \ldots, i$ gemacht werden kann

Berechnung eines Eintrags: Wir initialisieren: $DP[0] = 0$.

Sei $i \geq 1$ und $j(i)$ der grösste Index j, sodass $m_j \leq m_i - d$ ($j(i) = 0$, falls so ein Index nicht existiert). Dann haben wir:

$$DP[i] = \max\{DP[i-1], p_i + DP[j(i)]\}$$

Berechnungsreihenfolge: Wir berechnen die Elemente von links nach rechts.

Lesen des Resultats: Das Resultat befindet sich in DP[n]

Laufzeit: Um $DP[i]$ zu berechnen brauchen wir zunächst den Wert von $j(i)$. Diesen können wir mit binarySearch in $O(\log n)$ finden, weil die Distanzen zum Hauptbahnhof aufsteigend sortiert sind. Somit dauert die Berechnung eines Elements $O(\log n)$ und da wir $n + 1$ Elemente haben, haben wir eine Gesamtlaufzeit von $O(n * \log n)$.