Kapitel 7
Dynamisches Programmieren

1.2 Algorithmen-Paradigmen

Generelle Techniken zur Lösung großer Klassen von Problemstellungen

Greedy algorithms

- “gefräßiger, gieriger” Ansatz, Wahl des lokalen Optimums
- Beispiele: Prim und Kruskal’s Algorithmen für MSB, Dijkstra’s Algorithmus für kürzeste Wege

Divide-and-conquer algorithms

- schrittweises Zerlegen des Problems der Größe n in kleinere Teilprobleme
- Beispiele: Mergesort, Quicksort

Dynamic programming

- dynamischer sukkzessiver Aufbau der Lösung aus schon berechneten Teillösungen
- Beispiele: Fibonacci Zahlen, Transitive Hülle, Floyd-Warshall Algorithmus für kürzeste Wege, dieses Kapitel

Brute force

Ausprobieren aller möglichen Lösungen

Dynamisches Programmieren (DP)

universität wien

Dynamisches Programmieren kann man anwenden wenn:

- ein Problem aus voneinander abhängigen Unterproblemen besteht
- die Unterprobleme aus Unter-Unterprobleme besteht, die nur einmal geloest werden müssen und deren Lösung gespeichert werden kann (z.B. in einem Array)

- Beispiele heute:
  - Knapsack Problem
  - Sequence von Matrix Multiplikationen
  - Längste gemeinsame Untersequenz

Unterschiede Divide & Conquer versus Dynamisches Programmieren

universität wien

Divide & Conquer:

- Unabhängige Unter-Probleme
- (Oft) rekursive Lösungen

Dynamisches Programmieren:

- Abhängige Unter-Probleme
- Unterprobleme müssen die gleichen Unter-Unter-Probleme lösen:
  - Algorithmische Methode: Jedes Unter-Unter-Problem nur einmal lösen, und die Lösungen von Unter-Unter-Problemen in einer Tabelle gespeicher und für die Lösung übergeordnete Unter-Probleme verwendet.

Häufiges Anwendungsfeld von DP

Optimierungsproblem: Finde eine Lösung mit optimalen (maximalen oder minimalen) Wert.

Eine optimale Lösung, nicht *die* optimale Lösung: wenn es mehr als nur eine optimale Lösung gibt, kann eine beliebige von ihnen als Antwort gegeben werden.

Erforderliche Schritte des DP

1. Beschreibe die Struktur einer optimalen Lösung.
2. Definiere rekursiv den Wert einer optimalen Lösung.
3. Berechne den Wert einer optimalen Lösung in einer Bottom-up Methode
4. Berechne eine optimale Lösung von errechneten/gespeicherten Informationen.

Diese Vorlesung:

- 0/1 Rucksackproblem
- Matrix-Ketten-Multiplikations Problem
- Längste gemeinsame Untersequenz

0/1 Rucksackproblem

Input: ein Rucksack mit Kapazität W und eine Reihe von *n* Objekten, numeriert *1,2,…,n*. Jedes Objekt *i* hat ein Gewicht w_i und ein Gewinn v_i.

Output: Ein Vektor x = $[x_1, x_2, \ldots, x_n]$ in welchem

$x_i = 0$ wenn Objekt *i* nicht im Rucksack ist, und
$x_i = 1$ wenn es im Rucksack ist, und der die folgenden Bedingungen erfüllt:

$$\sum_{i=1}^{n} w_i x_i \leq W$$

(all Objekte passen in den Rucksack) und

$$\sum_{i=1}^{n} v_i x_i$$

ist maximiert

0/1 Rucksackproblem

Brute-force Algorithmus: Betrachte **alle** möglichen Teilmenge von *n* Objekten und wähle als Antwort die Teilmenge, welche in den Rucksack passt und den Gewinn maximiert.

$\Rightarrow$ Exponentielle Laufzeit in *n*

0/1 Rucksackproblem: DP Lösung

Lass $c_{i,w}$ der maximale Gewinn für einen Rucksack mit Kapazität *w* sein, welcher nur Objekte aus der Menge *{1,2,…,i} benutzt*.

Die DP Formulierung ist:

$$c_{i,w} = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c_{i-1,w} & \text{if } w_i > w \\ \max\{v_i + c_{i-1,w-w_i}, c_{i-1,w}\} & \text{if } i > 0 \text{ and } w_i \leq w \end{cases}$$

Erstelle eine Tabelle *F* mit der Größe *n x W+1* in einer zeilendominierenden Reihenfolge: Berechne zuerst die erste Zeile, dann die zweite, etc.

Ein Eintrag in einer Zeile benötigt zwei Einträge aus der vorherigen Zeile: einen von der selben Spalte und einen von einer Spalte mit kleinerem Gewicht

0/1 Rucksackproblem: DP Algorithmus

universität wien

```
1:  for w = 0...W do
2:      c_{0,w} ← 0
3:  end for
4:  for i = 1...n do
5:      c_{i,0} ← 0
6:      for w = 1...W do
7:          if w_i ≤ w then
8:              if v_i + c_{i-1,w-w_i} > c_{i-1,w} then
9:                  c_{iw} = v_i + c_{i-1,w-w_i}
10:             else
11:                 c_{iw} = c_{i-1,w}
12:             end if
13:         else
14:             c_{iw} = c_{i-1,w}
15:         end if
16:     end for
17: end for
18: return c_{nW}.
```

0/1 Rucksackproblem

Die Datenverarbeitung jedes Eintrages dauert gleich lang: ⇨ Laufzeit ist $\Theta(nW)$.

Es ist kein Algorithmus bekannt, welcher nur Zeit polynomial in n benötigt.

0/1 Rucksackproblem

Greedy Algorithmus: Ordne die Objekte nach ihrem $\frac{v_i}{w_i}$ Werten

und arbeite alle Objekte in dieser Reihenfolge ab: Wenn das derzeitige Objekt hinzugefügt werden kann, ohne dass W überschritten wird, füge das Objekt zum Rucksack hinzu.

Beispiel: W = 10

i	v_i	w_i
1	2	1
2	19	10

Greedy legt nur das erste Objekt in den Rucksack, dh der Wert ist 2, während die optimale Lösung 19 ist

→ Die Lösung produziert von Greedy kann beliebig schlecht sein

Matrix Multiplikation

Matrix: Eine $n \times m$ Matrix $\mathrm{A} = [a[i,j]]$ ist ein zweidimensionales Feld

$$\mathrm{A} = \begin{bmatrix} a[1,1] & a[1,2] \cdots & a[1,m-1] & a[1,m] \\ a[2,1] & a[2,2] \cdots & a[2,m-1] & a[2,m] \\ \vdots & \vdots & \vdots & \vdots \\ a[n,1] & a[n,2] \cdots & a[n,m-1] & a[n,m] \end{bmatrix},$$

welches n Reihen und m Spalten hat.

Beispiel: Folgend eine 4×5 Matrix:

$$\begin{bmatrix} 12 & 8 & 9 & 7 & 6 \\ 7 & 6 & 89 & 56 & 2 \\ 5 & 5 & 6 & 9 & 10 \\ 8 & 6 & 0 & -8 & -1 \end{bmatrix}.$$

Matrix Multiplikation (cont.)

Das Produkt $C = AB$ von einer $p \times q$ Matrix A und einer $q \times r$ Matrix wir angegeben mit

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

Für $1 \leq i \leq p$ und $1 \leq j \leq r$.

Beispiel:

Wenn $$A = \begin{bmatrix} 1 & 8 & 9 \\ 7 & 6 & -1 \\ 5 & 5 & 6 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 8 \\ 7 & 6 \\ 5 & 5 \end{bmatrix},$$

Dann $$C = AB = \begin{bmatrix} 102 & 101 \\ 44 & 87 \\ 70 & 100 \end{bmatrix}.$$

Matrix Multiplikation (cont.)

- Wenn AB definiert ist, kann BA nicht definiert sein.
- Es ist möglich das $AB \neq BA$.
- Die Multiplikation ist rekursiv definiert mit

$$A_1 A_2 A_3 \cdots A_{s-1} A_s = A_1(A_2(A_3 \cdots (A_{s-1} A_s))).$$

- Matrix Multiplikation ist assoziativ, z.B.:

$$A_1 A_2 A_3 = (A_1 A_2) A_3 = A_1 (A_2 A_3),$$

so verändern die Klammersetzung das Resultat nicht.

Direkte Matrix Multiplikation

Gibt man einer $p \times q$ Matrix A und einer $q \times r$ Matrix B, dann können alle mit dem direkten Weg der Multiplikation $C = AB$ berechnet werden

$$c[i,j] = \sum_{k=1}^{q} a[i,k]b[k,j]$$

Es gilt $1 \leq i \leq p \; und \; a \leq j \leq r$

Komplexität der Direkten Matrix Multiplikation

Nicht das C hat pr Einträge sondern jeder Eintrag benötigt Zeit $\theta\,(q)$ zum berechnen, daher wird für den ganzen Prozess Zeit benötigt. $\theta\,(pqr)$

Direkte Matrix Multiplikation von ABC

Gibt man einer $p \times q$ Matrix A, einer $q \times r$ Matrix B und einer $r \times s$ Matrix C
Dann kann ABC auf zwei Wege berechnet werden.
$(AB)C$ and $A(BC)$:

Die Mehrzahl der Multiplikationen benötigen:

$$mult[(AB)C] = pqr + prs = pr(q+s)$$
$$mult[A(BC)] = qrs + pqs = qs(r+p)$$

Wenn $p = 5, q = 6\ und\ s = 2, dann$

$$mult[(AB)C] = 180.$$
$$mult[A(BC)] = 88.$$

Ein sehr große Differenz!

⇨ Die Reihenfolge der Multiplikation (Klammersetzung) beeinflusst die Berechnungszeit

Matrix Ketten Multiplikation Problem

Seien $p_{0,}\, p_{1,\cdots,} p_n$
zu Matrix Sequenzen $A_{1,} A_2, \cdots A_n$
wo A_i die Größe $p_{i-1} \times\ p_1$ hat,
bestimmt die "Multiplikation Sequenz", dass die Anzahl der Skalar-Multiplikationen verringert wird bei einer Berechnung $A_{1,} A_2, \cdots A_n$.
In diesem Fall, bestimmt die Klammersetzung die Multiplikation.

$$A_1A_2A_3A_4 = (A_1A_2)(A_3A_4)$$
$$= (A_1(A_2(A_3A_4) = A_1((A_2A_3)A_4)$$
$$= ((A_1A_2)A_3)(A_4) = (A_1(A_2A_3))A_4)$$

Vollständige Suche: $\Omega(4^n\ /n^{3/2})$.

Step 1: Bestimmte die Struktur einer optimalen Lösung)
(In diesem Fall, mit Klammersetzung).

Zerlege das Problem in Unterprobleme:
Für jedes Paar $1 \leq i \leq j \leq n$, bestimme die Multiplikation Sequenz für$A_{i..j} = A_i A_{i+1} \cdots A_j$, dies minimiert die Anzahl der Multiplikationen.

Genauer: $A_{i..j}$ ist eine $p_{i-1} \times p_j$ Matrix.

Das ursprüngliche Problem:
Bestimme die Sequenz der Multiplikation für $A_{1..n}$.

Step 1: Bestimmte die Struktur einer optimalen Lösung)
(In diesem Fall, mit Klammersetzung).

High-Level Klammersetzung für $A_{i..j}$
Für jede optimale Multiplikationsequenz, multiplizieren im letzten Schritt die zwei Matrizen $A_{i..j}$ und $A_{k+1..j}$
um$A_{i..j}$ zu erhalten.

$$A_{i..j} = (A_i \cdots A_k)(A_{k+1} \cdots A_j) = A_{i..k}A_{k+1..j}.$$

Beispiel:

$$A_{3..6} = (A_3(A_4A_5))(A_6) = A_{3..5}A_{6..6}.$$

Hier gilt $k = 5$.

Entwicklung des DP Vorgehens

Step 1- Fortsetzung: Demzufolge ist das Problem der Ermittlung der optimalen Multiplikationssequenz in zwei Fragen aufgesplittet:

- Wie können wir entscheiden wo die Kette gebrochen ist (Was ist k)?

  (Suche alle möglichen Werte von k)

- Wie sollen wir in den Unter-Ketten $A_{i..k}$ und $A_{k+1..j}$ die Klammern setzen?

  (Das Problem besitzt optimale Unter-Strukturen,
  somit müssen $A_{i..k}$ und $A_{k+1..j}$ optimal sein,
  daher können wir dieselbe Prozedur rekursiv anwenden)

Optimale Unterstrukture Eigenschaft

universität wien

Optimale Unter-Struktur Eigenschaft: Wenn die optimale Lösung von $A_{i..j}$ die Aufsplitterung in $A_{i..k}$ und $A_{k+1..j}$ ist, dann muss die optimale Klammersetzung in $A_{i..j}$ auch eine optimale Klammersetzung in $A_{i..k}$ und in $A_{k+1..j}$ sein.

Wenn die Klammersetzung von $A_{i..k}$ nicht optimal war, können wir diese durch eine besser Klammersetzung ersetzen und bekommen so eine bessere Lösung für $A_{i..j}$, was zu einen Widerspruch führt.

Ähnlich auch bei einer nicht optimalen Klammersetzung von $A_{k+1..j}$, hier können wir auch eine bessere Klammersetzung einfügen und bekommen eine bessere Lösung für $A_{i..j}$, auch diese führt zu einem Widerspruch.

Step 2: Definiere rekursiv den Wert einer optimalen Lösung.

Wir speichern die Lösungen der Unterprobleme in einem Array:

Für $1 \leq i \leq j \leq n$, bezeichnet $m[i, j]$ die Mindestzahl von Multiplikationen, die man zum Berechnen von $A_{i..j}$ benötigt.

Die optimalen Anzahl von Multiplikationen können durch die folgende rekursive Definition bestimmt werden.

universität wien

Step 2: Definiere rekursiv den Wert einer optimalen Lösung.

$$m[i,j] = \begin{cases} 0 & i = j, \\ min_{i \le k < j} \left(m[i,k] + m\,[k+1,j] + p_{i-1}p_k p_j\right) & i < j \end{cases}$$

Beweis: Jede optimale Sequenz von der Multiplikation von $A_{i..j}$ ist äquivalent zu einer Auswahl von Aufsplittungen

$$A_{i..j} = A_{i..k}A_{k+1..j}$$

für jedes k, wobei die Multiplikationssequenzen von $A_{i..k}$ und $A_{k+1..j}$, auch optimal sind. Daher gilt:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

Step 2 - Fortsetzung: wir wissen, dass für jedes k folgendes gilt

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$$

Wir wissen nicht was k ist,

Es gibt nur $j - i$ mögliche Werte für k, so können wir sie alle überprüfen und den Wert finden, welcher zu kleinsten Kosten führt.

Daher:

$$m[i,j] = \begin{cases} 0 & i = j, \\ min_{i \le k < j} \left(m[i,k] + m\,[k+1,j] + p_{i-1}p_k p_j\right) & i < j \end{cases}$$

Step 3: Errechne den Wert einer optimalen Lösung in einer Bottom-Up Methode.

Unsere Tabelle: $m[1..n, 1..n]$.

$m[i, j]$ ist nur definiert für $i \leq j$.

Wenn wir die Gleichung

$$m[i, j] = \min_{i \leq k \leq j}(m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$$

zum Berechnen von $m[i, j]$ benutzen, müssen wir bereits $m[i, k]$ und $m[k + 1, j]$ ermittelt haben.

Für beide Fälle ist die entsprechende Länge der Matrix-Kette weniger als $j - i + 1$. Daher sollte der Algorithmus die Tabelle in aufsteigender Reihenfolge der Matrix-Ketten-Länge auffüllen.

In der Reihenfolge werden die Werte von *m* berechnet:

$m[1,2], m[2,3], m[3,4], \cdots, m[n-3, n-2], m[n-2, n-1], m[n-1, n]$

$m[1,3], m[2,4], m[3,5], \cdots, m[n-3, n-1],\ m[n-2, n]$

$m[1,4], m[2,5], m[3,6], \cdots, m[n-3, n]$

$\vdots$

$m[1, n-1], m\,[2, n]$

$m[1, n]$

MCM Algorithmus

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n                ▷ l is the chain length.
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                         then m[i, j] ← q
12                              s[i, j] ← k
13  return m and s
```

Quelle: Cormen, Leiserson, Rivest, Stein
Introduction to Algorithms

Dynamisches Programmieren: Warnhinweise

universität wien

Beim Entwicklung eines dynamischen Programmierungsalgorithmus gibt es zwei Teile:

1. Das Finden einer geeigneten optimalen Unterstruktur und einer entsprechenden Gleichung.

Bsp.:

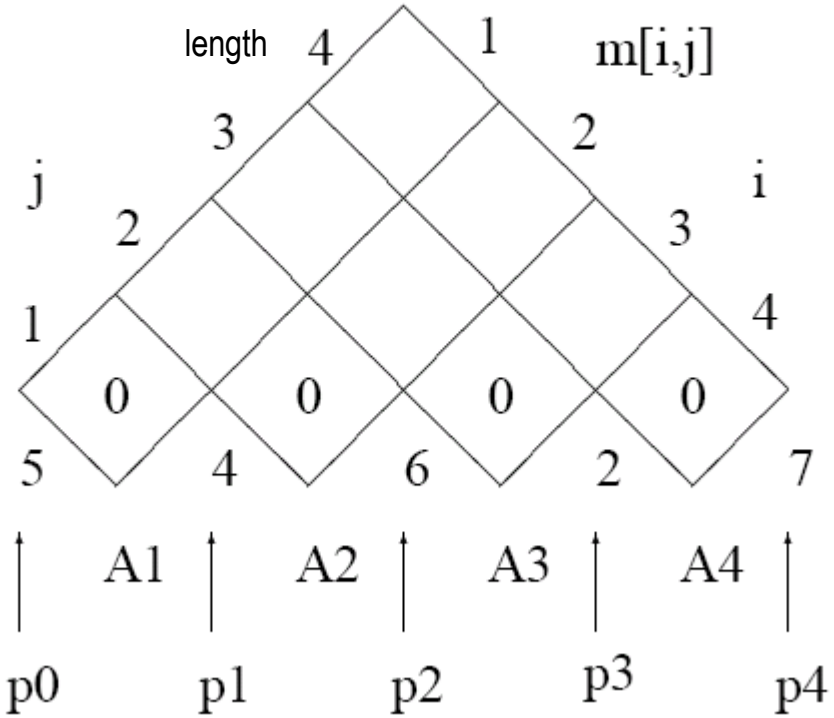
$$m[i,j] = \min_{i \le k < j} (m[i,k] + m]k+1,j] + p_{i-1}p_k p_j)$$

2. Die richtige Reihenfolge beim Ausfüllen der Tabelle.

Wenn ein Tabellenwert berechnet wird unter der Verwendung einer Gleichung, müssen alle Tabellenwerten der Gleichung bereits berechnet sein.

Beispiel: Gegeben ist eine Kette von vier Matrizen A_1, A_2, A_3 und A_4, mit $p_0 = 5, p_1 = 4, p_2 = 6, p_3 = 2$ und $p_4 = 7$.
Finde $m[1,4]$.
Anzahl der Multiplikationen für $(A_{1\ldots3})A_4$ist 88 + 5*2*7 = 158

S0: Initialization



Step 4: Konstruiere eine optimale Lösung von bereits berechneten Informationen – extrahiere die verwendete Sequenz.

Idee: Gegeben die Werte $s[1..n, 1..n]$, wo $s[i,j] = k$ die optimale Aufsplitterung der Berechnung $A_{i..j} = A_{i..k}A_{k+1..j}$ bezeichnet.

Die Werte $s[1..n, 1..n]$ können rekursiv benutzt werden, um die Multiplikationssequenz zu berechnen.

Wie berechnet man eine Multiplikationssequenz?

$$s\,[1,n] \qquad (A_1 \cdots A_{s[1,n]})(A_{s[1,n]+1} \cdots A_s)$$

$$s\,[1,s\,[1,n]] \qquad (A_1 \cdots A_{s[1,s[1,n]]})(A_{s[1,s]1,n]]+1} \cdots A_{s[1,n]})$$

$$s\,[s[1,n]+1,n] \qquad \left(A_{s[1,n]+1} \cdots A_{s[s[1,n]+1,n]}\right) \times$$

$$\vdots \qquad \left(A_{s[s[1,n]+1n]} \cdots A_n\right)$$

$$\vdots$$

Setze dies rekursiv fort bis die Multiplikationssequenz bestimmt ist.

Step 4:

universität wien

Erstelle eine optimale Lösung von den berechneten Informationen – extrahiere die aktuelle Sequenz.

Beispiel: Finden der Sequenzen-Multiplikation für n = 6. Gehe davon aus, dass das Feld s[1..6,1..6] berechnet ist. Die Sequenzen-Multiplikation ist folgendermaßen wiedergewonnen.

$s[1, 6] = 3$ $(A_1A_2A_3)(A_4A_5A_6)$
$s[1, 3] = 1$ $(A_1(A_2A_3))$
$s[4, 6] = 5$ $((A_4A_5)\,A_6)$

Somit lautet die schlussendliche Multiplikationssequenz

$(A_1(A_2A_3))\,((A_4A_5)\,A_6)$.

MCM Algorithmus

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1
 2  for i ← 1 to n
 3       do m[i, i] ← 0
 4  for l ← 2 to n                ▷ l is the chain length.
 5       do for i ← 1 to n − l + 1
 6            do j ← i + l − 1
 7               m[i, j] ← ∞
 8               for k ← i to j − 1
 9                  do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                     if q < m[i, j]
11                        then m[i, j] ← q
12                             s[i, j] ← k
13  return m and s
```

Quelle: Cormen, Leiserson, Rivest, Stein
Introduction to Algorithms

Komplexität

Laufzeit:

Verschachtelte Schleifen, jeder Schleifenindex übernimmt ≤ n Werte

Die Dauer zur Ausführung der innersten Schleife ist O(1)

⇨Zeitkomplexität $\Theta(n^3)$

Speicherplatzkomplexität: $\Theta(n^2)$

Tatsächliche Matrix Multiplikation

Das tatsächliche Multiplikationprogram benutzt $s[i, j]$ Wert, um zu bestimmen wie die Sequenz geteilt werden kann. Es wird angenommen, dass the i-te Matrix in einem Array $A[i]$ gespeichert ist und dass $s[i, j]$ von der vorherigen Prozedure *Matrix-Chain-Order* berechnet. Die Funktion *Mult* gibt eine Matrix zurück.

```
Mult(A, s, i, j)
{
    if (i < j)
    {
        X = Mult(A, s, i, s[i, j]);
            X is now A_i ··· A_k, where k is s[i, j]
        Y = Mult(A, s, s[i, j] + 1, j);
            Y is now A_{k+1} ··· A_j
        return X * Y;   multiply matrices X and Y
    }
    else return A[i];
}
```

Longest Common Subsequence (LCS)

universität wien

Anwendungsbereich: DNA Analysen

- DNA Stränge = Eine Sequenz von Symbolen A,C,G,T
- Gemeinsames Problem in der DNA Analyse: Vergleich von 2 DNA Strängen
  - S=ACCGGTCGAGCTTCGAAT
  - *X*=ACGCTAC
  - *Y*=CTGACA
- Messung der Ähnlichkeiten mittels: longest common subsequence

Unter-Sequenz von *X*: ist *X* mit einigen weggelassenen Symbolen.

- *Z*=CGTC ist eine Unter-Sequenz von *X*=ACGCTAC.

Gemeinsame Untersequenz *Z* von *X* und *Y*:

Eine Untersequenz von *X* und auch eine Untersequenz von*Y*.

- *Z*=CGA ist eine gemeinsame Untersequenz von *X*=ACGCTAC als auch von*Y*=CTGACA.

Longest Common Subsequence (LCS)

Longest Common Subsequence (LCS): Die Längste der gemeinsamen Untersequenzen.

Z' =CGCA ist die LCS von *X* und *Y*.

LCS Problem:

Input: 2 Sequenzen $X=<x_1, x_2,\ldots, x_m>$ und $Y=<y_1, y_2,\ldots, y_n>$

Output: LCS.

LCS Intuitive Lösung – brute force

Brute Force Algorithmus: Liste alle möglichen Unter-Sequenzen von X auf, kontrolliere ob diese auch Untersequenzen von Y sind, behalte jedes mal die Längere.

Laufdauer Analyse:

Jede Unter-Sequenz entspricht einer Teilmenge der Indizes {1,2,…,*m*}, es gibt 2^m Untersequenzen

Die Laufdauer ist exponential in *m*.

Notation

$X=<x_1, x_2,\ldots, x_m> =: X_m$

$X=<x_1, x_2,\ldots, x_i> =: X_i$

Gleiches gilt für *Y*

LCS Step 1: Optimale Unterstruktur

universität wien

Definiere die optimale Unter-Struktur von LCS.

Theorem: Gegeben $X=<x_1, x_2,\ldots, x_m>$ und $Y=<y_1, y_2,\ldots,y_n>$ dann ist $Z=<z_1, z_2,\ldots, z_k>$ eine LCS von X und Y,

1. wenn $x_m= y_n$, dann $z_k= x_m= y_n$, und Z_{k-1} ist die LCS von X_{m-1} und Y_{n-1}.
2. wenn $x_m \neq y_n$, dann

   $z_k \neq x_m$ bedeutet Z ist die LCS von X_{m-1} und Y_n.

   $z_k \neq y_n$ bedeutet Z is the LCS von X_m und Y_{n-1}.

LCS Step 2: Rekursive Lösung

Das Theorem besagt:

Wenn $x_m = y_n$, finde LCS von X_{m-1} und Y_{n-1} und füge dann x_m an.

wenn $x_m \neq y_n$, finde die LCS von X_{m-1} und Y_n und die LCS von X_m und Y_{n-1}, nimm die Längere.

Überlappende Unter-Strukturen:

Beide LCS von X_{m-1} und Y_n und LCS von X_m und Y_{n-1} können nützlich sein für die Lösung von LCS von X_{m-1} und Y_{n-1}.

$c[i,j]$ ist die Länge der LCS von X_i und Y_j .

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j, \end{cases}$$

LCS Step 3: Berechne die Länge der LCS

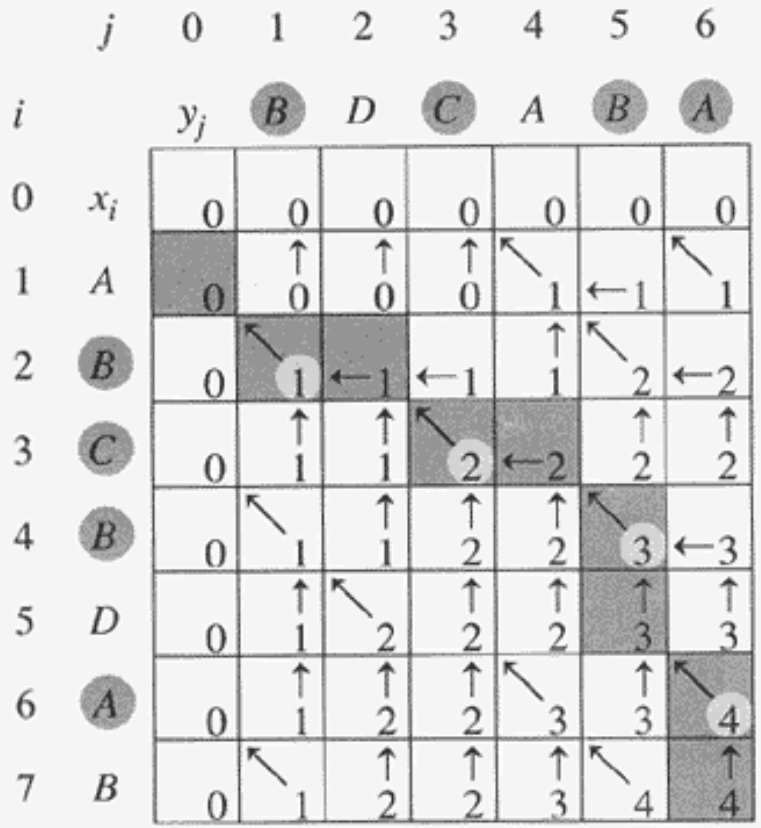
Algorithmen benutzen 2 Matrizen:

- $c[0..m,0..n]$, wie $c[i,j]$ oben definiert.
  - $c[m,n]$ ist die Antwort (Länge der LCS).
- $b[1..m,1..n]$, wie $b[i,j]$ verweist auf den Tabelleneintrag, welche der optimalen Lösung des Unter-Problem entspricht, die bei der Berechnung von $c[i,j]$ gewählt wurde.
  - Von $b[m,n]$ lässt sich die LCS finden.

LCS Dynamisches Programmieren Algorithmus


```
LCS-LENGTH(X, Y)
 1  m ← length[X]
 2  n ← length[Y]
 3  for i ← 1 to m
 4       do c[i, 0] ← 0
 5  for j ← 0 to n
 6       do c[0, j] ← 0
 7  for i ← 1 to m
 8       do for j ← 1 to n
 9            do if x_i = y_j
10                 then c[i, j] ← c[i − 1, j − 1] + 1
11                      b[i, j] ← "↖"
12                 else if c[i − 1, j] ≥ c[i, j − 1]
13                        then c[i, j] ← c[i − 1, j]
14                             b[i, j] ← "↑"
15                        else c[i, j] ← c[i, j − 1]
16                             b[i, j] ← "←"
17  return c and b
```

Quelle: Cormen, Leiserson, Rivest, Stein
Introduction to Algorithms



i		j	0	1	2	3	4	5	6
				y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Quelle: Cormen, Leiserson, Rivest, Stein
Introduction to Algorithms

Figure 15.6 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the path is shaded. Each "↖" on the path corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

LCS DP –Step 4: Konstruierung LCS

Call **Print-LCS**(b, X, |X|, |Y|)

```
PRINT-LCS(b, X, i, j)
1  if i = 0 or j = 0
2     then return
3  if b[i, j] = "↖"
4     then PRINT-LCS(b, X, i − 1, j − 1)
5          print x_i
6  elseif b[i, j] = "↑"
7     then PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

Quelle: Cormen, Leiserson, Rivest, Stein
Introduction to Algorithms

Dynamisches Programmieren Was nehmen wir mit?

universität wien

DP zwei wichtige Eigenschaften:

- Optimale Unterstruktureneigenschaft
- Überlappende Unter-Probleme

4 Steps des DP

Unterschiede zwischen den Divide-and-Conquer Algorithmen und den DP Algorithmen:

- Unabhängige und abhängige Unter-Probleme