Algorithmen und Datenstrukturen

Dynamische Programmierung

Prof. Justus Piater, Ph.D.

25. Mai 2025

Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch *Data Structures and Algorithms in Java* [Goodrich u. a. 2014].

Inhaltsverzeichnis

1 Matrix-Kettenmultiplikation **2**

2 Paradigma **4**

3 Längste gemeinsame Untersequenzen **8**

4 Epilog **16**

5 Literatur **16**

Einführung

Nehmen wir an, Sie möchten eine Kette von Matrizen miteinander multiplizieren. Matrix-Multiplikation ist assoziativ. Sie müssen also die Multiplikationen nicht unbedingt von links nach rechts durchführen. In welcher Reihenfolge sollten Sie nun multiplizieren? Diese Frage ist von praktischer Bedeutung, denn die Reihenfolge hat einen großen Einfluss auf die Anzahl der skalaren Multiplikationen, die Sie dabei insgesamt durchführen.

Fragen wir uns nun: Welche Multiplikation führen wir als *letztes* durch? Dies entspricht der Teilung der Matrix-Kettenmultiplikation in zwei Teilketten, deren Produkte am Schluss miteinander multipliziert werden. Könnten wir diese letzte Multiplikation optimal bestimmen, dann könnten wir rekursiv nach dem Teile-und-Herrsche-Prinzip fortfahren.

Wir wissen jedoch nicht, welche Multiplikation bei einer optimalen Lösung die letzte sein wird. Also prüfen wir für *jede* der Multiplikationen, wie effizient die jeweils bestmögliche Gesamtlösung wäre, und wählen die beste.

Dies können wir tun, wenn wir für jede der kürzeren Teilketten jeweils die optimale Lösung bereits kennen. Wir benötigen also die Lösungen für alle Teilketten von Matrix i bis Matrix j. Diese Lösungen legen wir in einer Tabelle ab, die nach i und j indiziert ist.

Die Lösungen für längere Teilketten können wir aus den Lösungen für kürzere Teilketten berechnen. Zur Lösung des Gesamtproblems müssen wir also lediglich die Tabelle füllen, indem wir den Wert jeder Zelle aus bereits berechneten Werten in anderen Zellen berechnen.

Dieses Verfahren nennt sich ***dynamic programming***. Dynamic-Programming-Algorithmen sind nicht immer einfach zu finden, aber sie sind oft effizient und sehr elegant und einfach implementierbar.

1 Matrix-Kettenmultiplikation

Matrix-Kettenmultiplikation [Slide 1]

$$A = BC \qquad A\colon d_0 \times d_2;\ B\colon d_0 \times d_1;\ C\colon d_1 \times d_2$$

$$a_{ij} = \sum_{k=0}^{d_1-1} b_{ik} c_{kj} \qquad \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & b_{ik} & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & c_{kj} & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & a_{ij} & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

$$A = A_0 A_1 \cdots A_{n-1} \qquad A_i\colon d_i \times d_{i+1}$$

Matrix-Multiplikation ist *assoziativ*.

Seien B 2×10, C 10×50, und D 50×20-Matrizen. Wie viele skalare Multiplikationen erfordern $(BC)D$ und $B(CD)$?

$10\cdot2\cdot50 + 50\cdot2\cdot20 = 3000$

$50\cdot10\cdot20 + 10\cdot2\cdot20 = 10400$

Wie finden wir die Klammerung, die die Anzahl der skalaren Multiplikationen minimiert?

Alle Klammerungen aufzählen? Deren Zahl ist exponentiell in n (ohne Beweis).

Stuktur des Problems [Slide 2]

- Das Problem besteht aus ***Unterproblemen*** $A_i A_{i+1} \cdots A_j$; sei N_{ij} deren Anzahl Multiplikationen. Das Gesamtproblem ist die Bestimmung von $N_{0,n-1}$.

  Diese Einsicht hilft uns jedoch nicht, das Gesamtproblem in getrennte Unterprobleme aufzuspalten (um einen Teile-und-Herrsche-Algorithmus zu entwickeln). Wir wissen nämlich nicht, *wo* wir es für eine optimale Gesamtlösung aufspalten müssen (d.h., wie wir im folgenden Punkt k wählen müssen).

- ***Optimalität der Unterprobleme***: Für jede optimale Lösung $A_i \cdots A_j = (A_i \cdots A_k)(A_{k+1} \cdots A_j)$ müssen beide Teillösungen jeweils ebenfalls optimal sein.

  Beweis?

Idee: Löse $A_i \cdots A_j$ durch Suche nach dem optimalen k:

$$N_{ij} = \min_{i \le k < j} \left\{ N_{ik} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \right\}$$

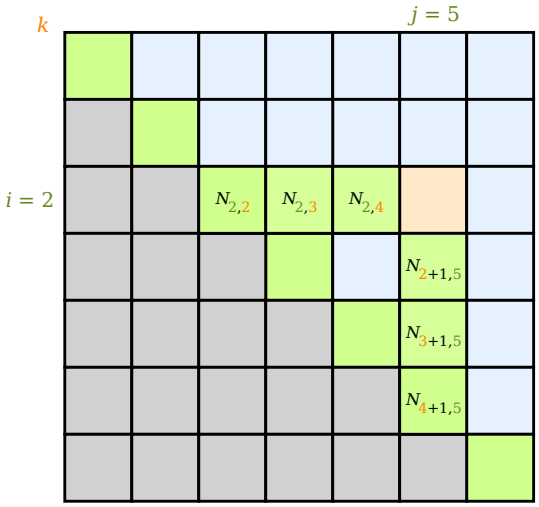
Wichtig

Überlappende Unterprobleme: Optimale Teillösungen für verschiedene k enthalten ihrerseits u.a. dieselben Teillösungen.

Berechnen wir *gemeinsame Teillösungen* nur einmal, können wir eine gewaltige Aufwandsreduktion gegenüber der Brute-Force-Lösung erwarten!

Algorithmus: Systematisches Ausfüllen der Tabelle N, immer auf Basis bereits ausgefüllter Werte (von der Diagonale nach oben rechts)

Tabelle: $N_{ij} = \min\limits_{i \le k < j} \left\{ N_{ik} + N_{k+1,j} + d_i d_{k+1} d_{j+1} \right\}$ **[Slide 3]**



Algorithmus [Slide 4]

1. Setze alle $N_{ii} = 0$.
2. Berechne alle $N_{i,j+1}$ parallel zur Diagonalen.
3. Iteriere Schritt 2.

```
Algorithm matrixChain(d_0, ..., d_n):
  Require: first dimensions d_0, ..., d_{n-1} of matrices A_0, ..., A_{n-1},
                   last dimension d_n of A_{n-1}.
  Ensure: N is an n × n table with N_ij the number of multiplications
                   required to compute A_i A_{i+1} ··· A_j.
  N ← n × n table with all-zero diagonal
  for b ← 1 to n − 1 do // # matrix products in subchain
    for i ← 0 to n − b − 1 do // start of subchain
      j ← i + b          // end of subchain
      N_ij ← ∞
      for k ← i to j − 1 do
        N_ij ← min{N_ij, N_ik + N_{k+1,j} + d_i d_{k+1} d_{j+1}}
  return N
```

Übungen:

- Bestimmen Sie die asymptotische Laufzeit dieses Algorithmus, und begründen Sie diese detailliert!
- Entwickeln Sie einen Algorithmus, der die tatsächliche Reihenfolge der Multiplikationen in geeigneter Form ausgibt, z.B. als vollständig geklammerten Ausdruck. (Tipp: Verwenden Sie eine $n \times n$ Hilfstabelle.)

Quiz [Slide 5]

Was ist die Laufzeit unseres `matrixChain()`-Algorithmus?

A: $\Theta(n^2 \log n)$

B: $\Theta(n^3)$

C: $\Theta(n^3 \log n)$

D: weiß nicht

2 Paradigma

Video 1 beginnt hier.

Fibonacci: Rekursion (ineffizient) [Slide 6]

$F_0 = 0$, $F_1 = 1$, $F_k = F_{k-2} + F_{k-1}$

```
Algorithm Fibonacci(k):
  Require: Integer k ≥ 0.
  Ensure: Return the k−th Fibonacci number.
  if k ≤ 1 then
    return k
  return Fibonacci(k − 2) + Fibonacci(k − 1)
```

Beispiel

Um die dynamische Programmierung zu motivieren, schauen wir uns einmal die gute alte Fibonacci-Folge an. Hier sehen wir ihre rekursive Definition und deren umittelbare Umsetzung in einen rekursiven Algorithmus: Eine Fibonacci-Zahl ist die Summe der beiden vorhergehenden Fibonacci-Zahlen. Die Rekursion fußt auf den ersten beiden Fibonacci-Zahlen 0 und 1.

Dieser rekursive Algorithmus ist extrem ineffizient, denn bei der Berechnung von `Fibonacci()` werden vorhergehende Fibonacci-Zahlen immer öfter wiederholt berechnet. `Fibonacci(`k`)` ruft rekursiv `Fibonacci(`$k-2$`)` und `Fibonacci(`$k-1$`)` auf, und dieser Aufruf von `Fibonacci(`$k-1$`)` ruft ebenfalls u.a. `Fibonacci(`$k-2$`)` auf. In jedem 2. rekursiven Aufruf wird also der gesamte 1. Aufruf erneut berechnet, und diese Verdoppelung findet bei *jedem* rekursiven Aufruf statt!

Dieses Problem tritt hier auf, weil wir Rekursion naiv verwenden. Der Algorithmus sieht nach Teile-und-Herrsche aus, ist es aber nicht, denn das Problem wird nicht in zwei voneinander unabhängige Teilprobleme aufgeteilt. Im Gegenteil, diese Teilprobleme ***überlappen*** sich massiv, und zwar in der Form wiederholt berechneter Fibonacci-Zahlen.

Dynamische Programmierung vermeidet solche Mehrfachberechnungen überlappender Unterprobleme.

Fibonacci: Rekursion mit Memoisation (effizient) [Slide 7]

```
Algorithm Fibonacci(k):
  Require: integer k ≥ 0.
  Ensure: the k−th Fibonacci number.
  if known[k] ≠ 0 then
    return known[k]
  if k ≤ 1 then
    return k
  known[k] ← Fibonacci(k − 2) + Fibonacci(k − 1)
  return known[k]
```

Bei Rekursion kann ***Memoisation*** verwendet werden, um redundante Mehrfachberechnungen zu vermeiden. Dies entspricht oft dem sogenannten *top-down dynamic programming*.

Beispiel

Wir können diese Mehrfachberechnungen auf einfache Weise vermeiden, indem wir uns jedes berechnete Teilergebnis merken: Wir schreiben uns ein Memo. Wenn ein Teilergebnis dann ein weiteres Mal angefordert wird, dann geben wir einfach das bereits berechnete Ergebnis zurück, anstatt es erneut zu berechnen. Hiermit schneiden wir effektiv den rekursiven Aufrufbaum an dieser Stelle nach unten hin komplett ab.

Solches Zwischenspeichern zur Vermeidung von Mehrfachberechnungen nennt sich ***Memoisation***.

Rekursion mit Memoisation angewendet auf überlappende Unterprobleme wird auch als ***Top-Down Dynamic Programming*** bezeichnet.

Dynamische Programmierung im eigentlichen Sinne arbeitet dagegen *bottom-up*, iterativ statt rekursiv, durch Ausfüllen einer Tabelle.

Fibonacci: Dynamische Programmierung [Slide 8]

```
Algorithm Fibonacci(k):
  Require: integer k ≥ 0.
  Ensure: the k−th Fibonacci number.
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to k do
    F[i] ← F[i − 2] + F[i − 1]
  return F[k]
```

Echte dynamische Programmierung vermeidet redundante Mehrfachberechnungen durch iteratives Ausfüllen einer Tabelle (*bottom-up*).

Beispiel

Hier sehen wir einen *Dynamic-Programming*-Algorithmus zur Berechnung von Fibonacci-Zahlen. Statt Mehrfachberechnungen zu erkennen und darauf zu reagieren, vermeidet er überhaupt das Auftreten von Mehrfachberechnungen, indem die Unterprobleme von unten her gelöst werden.

Er führt also `Fibonacci(k)` nicht rekursiv auf seine Vorgänger `Fibonacci(k − 2)` und `Fibonacci(k − 1)` zurück, sondern beginnt mit $F[0]$ und $F[1]$, und baut die nachfolgenden Fibonacci-Zahlen iterativ darauf auf.

Dies geschieht durch Ausfüllen der Tabelle F: $F[i]$ ist die Summe der beiden bereits berechneten Felder $F[i-2]$ und $F[i-1]$.

Damit ist die Laufzeit von `Fibonacci(k)` *linear* in k, eine erhebliche Verbesserung zum naiven rekursiven Algorithmus, der exponentiell in k ist.

Der Platzbedarf ist in dieser einfachen Version jedoch ebenso wie beim rekursiven Algorithmus linear in k.

Falls wir nur am Endergebnis `Fibonacci(`k`)` interessiert sind und nicht an den vorhergehenden Fibonacci-Zahlen, dann können wir diese Letzteren verwerfen, sobald sie nicht mehr gebraucht werden. Wir können den Algorithmus auf einfache Weise so abändern, dass anstelle der kompletten Tabelle $F[k]$ lediglich die beiden vorhergehenden Werte gespeichert und immer wieder überschrieben werden. Damit können wir seinen Platzbedarf konstant halten.

Algorithmisches Paradigma: *Dynamische Programmierung (Dynamic Programming)* [Slide 9]

Teile-und-Herrsche-Algorithmen teilen ein Problem in *disjunkte* Unterprobleme auf, die dann rekursiv *unabhängig voneinander* gelöst werden.

Viele Probleme lassen sich nicht vorab in disjunkte Unterprobleme aufteilen, und sind doch auf nützliche Weise zerlegbar:

- Das Problem lässt sich auf gleichartige ***Unterprobleme*** zurückführen, die aufeinander aufbauen.
- ***Optimalität der Unterprobleme***: Jede optimale Lösung eines (Unter-)Problems lässt sich aus *optimalen* Lösungen seiner Unterprobleme berechnen.
- ***Überlappende Unterprobleme***: In der hierarchischen Zerlegung des Gesamtproblems treten dieselben Unterprobleme mehrfach auf.

  Diese Eigenschaft macht Teile-und-Herrsche ineffizient oder unmöglich, aber wird durch dynamische Programmierung explizit genutzt.

Dynamische Programmierung verwendet eine ***Tabelle***, um die Lösungen der Unterprobleme zu speichern. Diese Tabelle wird ***iterativ*** ausgefüllt, indem für die Lösung jedes Unterproblems auf Lösungen bereits gelöster Unterprobleme zurückgegriffen wird.

Definition

Das algorithmische Paradigma der ***dynamischen Programmierung*** lässt sich auf Probleme anwenden, die die folgenden drei Eigenschaften besitzen:

1. Das Problem lässt sich auf gleichartige ***Unterprobleme*** zurückführen, die aufeinander aufbauen.
2. Die Eigenschaft der ***Optimalität der Unterprobleme*** besagt, dass sich jedes Unterproblem und damit auch das Gesamtproblem optimal lösen lässt, indem man auf optimalen Lösungen eines oder mehrerer Unterprobleme aufbaut.

Diese beiden Eigenschaften sind zunächst nichts überwältigend Besonderes. Viele Probleme besitzen eine solche Struktur, und viele solche Probleme lassen sich z.B. mit Teile-und-Herrsche-Algorithmen lösen. Aber jetzt kommt die 3. Eigenschaft:

3. Die Unterprobleme ***überlappen*** sich. Das kann bedeuten, dass in der hierarchischen Zerlegung des Gesamtproblems dieselben Unterprobleme immer wieder auftauchen. Oder es gibt mehrere mögliche hierarchische Zerlegungen, und es müssen im Prinzip alle durchprobiert werden, und dabei treten immer wieder dieselben Unterprobleme auf.

Es ist insbesondere die 3. Eigenschaft, die die dynamische Programmierung von Teile-und-Herrsche-Algorithmen absetzt. Teile-und-Herrsche ist nur anwendbar, wenn das Problem in *voneinander unabhängige* Teilprobleme zerlegbar ist.

Dynamische Programmierung ist anwendbar, wenn eine solche Zerlegung in disjunkte Teilprobleme nicht möglich ist. Sie baut ebenfalls auf gleichartigen Unterproblemen auf, nutzt aber Überlappungen zwischen den Unterproblemen explizit aus.

Bei der dynamischen Programmierung wird typischerweise eine Tabelle angelegt, in der jedes Feld einem Unterproblem entspricht. Eines oder mehrere dieser Felder entsprechen dem Gesamtproblem.

Diese Tabelle wird nun *iterativ* mit Lösungen dieser Unterprobleme befüllt.

Für die Lösung eines gegebenen Unterproblems wird hierbei auf eine oder mehrere Lösungen zurückgegriffen, die bereits berechnet und in den entsprechenden Feldern der Tabelle gespeichert sind.

Oft sind die berechneten Lösungen jedes Unterproblems unmittelbar optimal. Es kommt aber auch vor, dass sie approximativ sind und erst nach vielen Iterationen über die Tabelle, so genannten *Sweeps*, zum Optimum konvergieren.

Am Ende stellen jedenfalls alle berechneten Werte eine optimale Lösung ihres jeweiligen Unterproblems dar, inklusive desjenigen Wertes oder derjenigen Werte, die die Gesamtlösung repräsentieren.

Dieses iterative Ausfüllen der Tabelle gab der dynamischen Programmierung ihren Namen. *Programmierung* hat hier nichts mit dem Schreiben von Computer-Code zu tun. Man sollte hier eher an die Zusammenstellung eines Fernsehprogramms denken.

Das Sendeprogramm des Tages können wir uns als Tabelle vorstellen. Jede Zeile entspricht einer Sendung oder einem Beitrag. Diese Tabelle gilt es auf eine Weise auszufüllen, die das Programm als Ganzes möglichst attraktiv macht. Um dies zu erreichen, müssen jedoch die unterschiedlichen Beiträge aufeinander abgestimmt sein. Eine Änderung an einer Stelle kann weitere Änderungen an anderer Stelle erfordern. Das macht die Programmplanung zu einem höchst dynamischen Prozess.

Nehmen Sie diese Deutung des Begriffs *dynamische Programmierung* allerdings nur als eine hilfreiche Anekdote; sein genauer Ursprung ist unbekannt.

3 Längste gemeinsame Untersequenzen

Längste gemeinsame Untersequenzen [Slide 10]

Video 2 beginnt hier.

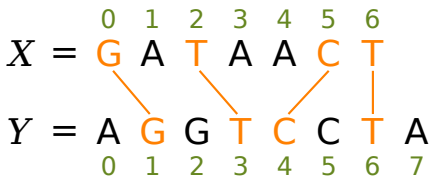
Eine ***Untersequenz*** einer Sequenz $X = x_0 x_1 \cdots x_{n-1}$ ist eine Sequenz S der Form $x_{i_1} x_{i_2} \cdots x_{i_k}$ mit $i_j < i_{j+1}$.

Die Zeichen von S finden sich in derselben Reihenfolge in X, aber nicht unbedingt lückenlos.

Wie finden wir eine längste gemeinsame Untersequenz einer Sequenz X der Länge n und einer Sequenz Y der Länge m?

Alle Untersequenzen von X aufzählen? Deren Zahl ist exponentiell in n, da jedes Zeichen entweder auftritt oder nicht.

Beispiel



Hervorgehoben ist die längste gemeinsame Untersequenz `GTCT`. Von dieser existieren in Y 4 verschiedene Instanzen, die sich in der Verwendung des ersten oder zweiten `G` bzw. `C` unterscheiden.

Darüber hinaus gibt es noch die längste gemeinsame Untersequenz `ATCT`, mit 2 Instanzen in Y.

Übung: Schreiben Sie einen Algorithmus, der in linearer Zeit nachprüft, ob eine *gegebene* Sequenz S eine Untersequenz einer beliebigen Sequenz X ist.

Erklärung

Betrachten wir nun ein klassisches Problem mit großer praktischer Bedeutung, das die dynamische Programmierung illustriert. Dies ist das Problem der *längsten gemeinsamen Untersequenzen.*

Eine *Untersequenz* einer Sequenz X ist eine Sequenz S, deren Elemente in derselben Reihenfolge auch in X vorkommen. Dazwischen können sich in X jedoch beliebige weitere Elemente befinden. S muss also nicht zusammenhängend in X vorkommen.

Angenommen, jemand gibt uns zwei verschiedene Sequenzen X und Y, und fragt uns nach einer längsten Sequenz S, die eine Untersequenz sowohl von X als auch von Y ist. Dies ist das Problem der längsten gemeinsamen Untersequenzen.

Dieses Problem zeigt sich in vielen praktischen Anwendungen. Typische Fälle finden wir in der Bioinformatik. Bei den beiden Sequenzen kann es sich beispielsweise um Basen eines DNA-Strangs handeln, und Biologen möchten bestimmte Gene miteinander vergleichen oder Gene und ihre Varianten nachverfolgen.

Hier sehen wir ein Beispiel. Die Zeichen `A`, `T`, `C` und `G` stehen hier für die Basen Adenin, Thymin, Cytosin und Guanin.

Eine längste gemeinsame Untersequenz dieser beiden DNA-Stränge ist `GTCT`; diese ist hier hervorgehoben.

Die dargestellte Paarung ist jedoch nicht die einzige. Alternativ könnten wir x_0 =`G` mit y_2 statt mit y_1 paaren, und x_5 =`C` mit y_5 statt mit y_4. Ingesamt existieren von dieser längsten gemeinsamen Untersequenz `GTCT` also 4 verschiedene Instanzen in Y, und eine in X.

Darüber hinaus können wir statt der ersten `G`s die ersten `A`s paaren. `ATCT` ist also ebenfalls eine längste gemeinsame Untersequenz.

Für eine *gegebene* Untersequenz S können wir mittels eines einfachen, gierigen Algorithmus leicht in linearer Zeit nachprüfen, ob eine Sequenz X sie als Untersequenz enthält. Wie aber können wir eine längste Untersequenz zweier Sequenzen X und Y finden?

Ein *Brute-Force*-Ansatz würde sämtliche Untersequenzen von X aufzählen und jeweils nachprüfen, ob sie auch eine Untersequenz von Y darstellen. Eine Sequenz der Länge n hat jedoch 2^n Untersequenzen, da jedes ihrer Elemente entweder in der Untersequenz vorhanden ist oder nicht.

Mit dynamischer Programmierung können wir unser Problem sehr viel effizienter lösen.

Unterprobleme [Slide 11]

- Das Problem besteht aus den ***Unterproblemen*** der Präfixe der beiden Sequenzen: Sei L_{jk} die Länge der längsten gemeinsamen Untersequenz von $x_0 \cdots x_{j-1}$ und $y_0 \cdots y_{k-1}$. Das Gesamtproblem ist dann die Bestimmung von L_{nm}.

  L_{jk} hält den Wert der *Zielfunktion* (die *Länge* der längsten gemeinsamen Untersequenzen) dieses Unterproblems.

- ***Optimalität der Unterprobleme***

  - Die längsten gemeinsamen Untersequenzen zweier Sequenzen Xc und Yc, die mit *demselben* Zeichen c enden, enthalten längste gemeinsame Untersequenzen von X und Y.
  - Die längsten gemeinsamen Untersequenzen zweier Sequenzen Xd und Ye, die mit *verschiedenen* Zeichen d bzw. e enden, sind jeweils mit längsten gemeinsamen Untersequenzen entweder von X und Ye oder von Xd und Y identisch.

- ***Überlappende Unterprobleme***: Dieselben Untersequenzen können Teil verschiedener, längerer Präfixe sein.

Erklärung

Der erste Schritt zu einem *Dynamic-Programming*-Algorithmus ist eine geeignete Formulierung des Gesamtproblems auf der Basis gleichartiger Unterprobleme. Eine solche Formulierung zu finden, erfordert oft etwas Kreativität.

Hier geht es um die *längsten* gemeinsamen Untersequenzen. Wir möchten also die *Länge* der gemeinsamen Untersequenzen maximieren. Das ist unsere *Zielfunktion*. Daher stürzen wir uns zunächst ausschließlich auf die Berechnung dieser maximalen Längen, und vertagen die Bestimmung der eigentlichen Untersequenzen auf später.

Das Problem der Bestimmung der Längen der längsten gemeinsamen Untersequenzen lässt sich nun auf recht einfache Weise auf Unterprobleme zurückführen. Angenommen, wir haben zwei Sequenzen X der Länge j und Y der Länge k, und wir kennen deren längste gemeinsame Untersequenzen bereits. Nun verlängern wir beide Sequenzen um *dasselbe* Zeichen. Hilft uns unsere Kenntnis der längsten gemeinsamen Untersequenzen der alten Versionen von X und Y bei der Bestimmung der neuen, längsten gemeinsamen Untersequenzen?

Ja, das tut sie in der Tat. Offensichtlich verlängern sich alle längsten gemeinsamen Untersequenzen der alten Versionen von X und Y um dieses neue Zeichen.

Auf Basis dieser Erkenntnis können wir nun die drei charakteristischen Eigenschaften der dynamischen Programmierung für das Problem der längsten gemeinsamen Untersequenzen ausarbeiten.

1. lässt sich unser Gesamtproblem auf der Basis gleichartiger *Unterprobleme* formulieren, nämlich der Bestimmung der längsten gemeinsamen Untersequenzen *kürzerer* Versionen der vollen Sequenzen, also deren *Präfixe*. Wir beginnen daher mit der Suche nach den längsten gemeinsamen Untersequenzen der *Präfixe* der beiden vollen Sequenzen.

   Die Werte der Zielfunktion der Lösungen unserer Unterprobleme, also die Längen der längsten gemeinsamen Untersequenzen der Präfixe, notieren wir in einer Tabelle L. Der Eintrag L_{jk} ist die Länge der längsten gemeinsamen Untersequenzen des Präfix der Länge j von X und des Präfix der Länge k von Y. Damit ist L_{nm} die Länge der längsten gemeinsamen Untersequenzen der vollen Sequenzen X und Y.

2. besitzt diese Formulierung die Eigenschaft der *Optimalität der Unterprobleme*, denn die längsten gemeinsamen Untersequenzen von X und Y bleiben Teil der längsten gemeinsamen Untersequenzen, wenn ich X und Y jeweils um dasselbe Zeichen verlängere.

   Auch für zwei Sequenzen, die mit *verschiedenen* Zeichen enden, gibt es eine optimale Unterproblemstruktur. Nur eines der beiden letzten Zeichen kann Teil einer gegebenen gemeinsamen Untersequenz sein. Daher kann ich entweder das eine oder das andere dieser beiden letzten Zeichen entfernen, und alle gemeinsamen Untersequenzen, die dieses Zeichen nicht enhalten, bleiben intakt.

3. *überlappen* diese Unterprobleme, denn die gemeinsamen Untersequenzen verschieden langer Präfixe können einander ganz oder auch nur teilweise enthalten. Dies verhindert sowohl die Aufspaltung in voneinander unabhängige Unterprobleme z.B. für einen Teile-und-Herrsche-Algorithmus, als auch die Lösung durch einen gierigen Algorithmus.

   Diese überlappenden Teillösungen werden durch einen *Brute-Force*-Algorithmus immer wieder neu berechnet. Dynamische Programmierung hingegen löst sie jeweils nur ein einziges Mal und baut die Lösungen größerer Unterprobleme darauf auf.

Damit haben wir unser Gesamtproblem auf der Basis von Unterproblemen mit Optimalitätseigenschaft formuliert, und eine Tabelle L definiert, mittels derer wir deren Lösungen berechnen und speichern.

Berechnung der Werte in der Tabelle L_{jk} [Slide 12]

Video 3 beginnt hier.

1. Setze alle $L_{j,0} = L_{0,k} = 0$.
2. (a) Falls $x_{j-1} = y_{k-1}$, dann gehört das gemeinsame, letzte Zeichen zu einer längsten gemeinsamen Untersequenz von $x_0 \cdots x_{j-1}$ und $y_0 \cdots y_{k-1}$. Daher setze $L_{jk} = 1 + L_{j-1,k-1}$.

   $$\begin{array}{cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ X = & G & A & T & A & A & C \\ Y = & A & G & G & T & C & \\ & 0 & 1 & 2 & 3 & 4 & \end{array} \qquad L_{6,5} = 1 + L_{5,4}$$

   Wir können dieses gemeinsame, letzte Zeichen also an die längste gemeinsame Untersequenz von $x_0 \cdots x_{j-2}$ und $y_0 \cdots y_{k-2}$ der Länge $L_{j-1,k-1}$ anfügen.

   (b) Falls $x_{j-1} \neq y_{k-1}$, ist es unmöglich, dass eine gemeinsame Untersequenz mit x_{j-1} *und* y_{k-1} endet. Daher setze $L_{jk} = \max\{L_{j-1,k}, L_{j,k-1}\}$.

   $$\begin{array}{ccccc} & 0 & 1 & 2 & 3 & 4 \\ X = & G & A & T & A & A \\ Y = & A & G & G & T & \\ & 0 & 1 & 2 & 3 & \end{array} \qquad L_{5,4} = \max\{L_{5,3}, L_{4,4}\}$$

   x_{j-1} oder y_{k-1} können jeweils Teil einer längsten gemeinsamen Untersequenz sein, aber nicht derselben. Man kann also x_{j-1} oder y_{k-1} (oder beide) entfernen, sodass mindestens eine längste gemeinsame Untersequenz intakt bleibt (die auf dem jeweils anderen letzten Zeichen oder vorher endet). Die Länge der *längsten* gemeinsamen Untersequenz dieser beiden Präfixe ist also die längere dieser beiden Möglichkeiten.

Erklärung

Als Nächstes müssen wir festlegen, wie wir unsere Tabelle L befüllen, indem jeder Eintrag auf Basis bereits berechneter Einträge berechnet wird.

Zunächst stellen wir fest, dass die Länge der längsten gemeinsamen Untersequenzen gleich Null ist, wenn mindestens eine der beiden Sequenzen leer ist. Wir können in L also alle Einträge der ersten Spalte und der ersten Zeile auf Null setzen.

Nun berechnen wir die restlichen Einträge unter Nutzung bereits vorhandener Einträge. Wie berechnen wir L_{jk} auf der Basis kürzerer Präfixe der Länge $j-1$ bzw. $k-1$?

Zur Erinnerung: L_{jk} ist die Länge der längsten gemeinsamen Untersequenzen des Präfix der Länge j von X und des Präfix der Länge k von Y. Vergleichen wir also das *letzte* Zeichen dieser beiden Präfixe miteinander, also x_{j-1} und y_{k-1}.

Sind diese identisch, dann ist dieses gemeinsame Zeichen Teil einer längsten gemeinsamen Untersequenz beider Präfixe, denn es verlängert deren bisher längste gemeinsame Untersequenzen um dieses gemeinsame Zeichen. Wir setzen also L_{jk} auf 1 plus die Länge der längsten gemeinsamen Untersequenzen der beiden um dieses Zeichen verkürzten Präfixe, also $1 + L_{j-1,k-1}$.

In unserer Illustration sehen wir die beiden aktuellen Präfixe der Länge $j = 6$ von X und der Länge $k = 5$ von Y, nämlich `GATAAC` und `AGGTC`.

Wir betrachten das jeweils letzte Zeichen dieser gezeigten Präfixe, also x_5 und y_4. Beide sind identisch, also Teil einer längsten gemeinsamen Untersequenz dieser beiden Präfixe, die sich damit um 1 verlängert gegenüber den längsten gemeinsamen Untersequenzen der um dieses letzte Zeichen verkürzten Präfixe. $L_{6,5}$ wird also auf $1 + L_{5,4}$ gesetzt.

Sind die jeweils letzten Zeichen der beiden aktuell betrachteten Präfixe jedoch nicht identisch, dann ist es unmöglich, dass *beide* Teil einer gemeinsamen Untersequenz dieser

Präfixe sind. Die längste gemeinsame Untersequenz kann sich also nicht um dieses Zeichen verlängern.

Es ist jedoch durchaus möglich, dass *eines* dieser jeweils letzten Zeichen Teil einer gemeinsamen Untersequenz ist, wie in diesem Beispiel $y_3 = \texttt{T}$. Die Länge L_{jk} der längsten gemeinsamen Untersequenzen dieser beiden Präfixe ist also gleich der Länge der längsten gemeinsamen Untersequenzen der beiden Präfixe *unter Ausschluss* des letzten Zeichens *eines* dieser beiden Präfixe, also gleich dem Maximum von $L_{j-1,k}$ und $L_{j,k-1}$. Dies sind die beiden Tabelleneinträge links und über dem aktuellen Eintrag L_{jk}.

Können wir die Tabelle so befüllen, dass für jedes Feld die benötigten Werte anderer Felder bereits vorhanden sind? Ja, das ist möglich. Zur Berechnung von L_{jk} benötigen wir die Einträge links und darüber, einschließlich des diagonalen Nachbarn. Diese Voraussetzung ist zu Beginn für $L_{1,1}$ erfüllt. Nach Berechnung von $L_{1,1}$ ist sie für seinen rechten und seinen unteren Nachbarn erfüllt, und so weiter. Es bietet sich also z.B. an, die Tabelle Zeile für Zeile auszufüllen.

Algorithmus: Berechnung der Tabelle L [Slide 13]

```
Algorithm LCS(X, Y):
  Require: Sequences X = x_0 ··· x_{n-1} and Y = y_0 ··· y_{m-1}.
  Ensure:  L is a table with L_jk the length of
                 an LCS of x_0 ··· x_{j-1} and y_0 ··· y_{k-1}.
  L ← (n+1) × (m+1) table with all-zero 1st column and row
  for j ← 1 to n do
    for k ← 1 to m do
      if x_{j-1} = y_{k-1} then
        L_jk ← L_{j-1,k-1} + 1
      else
        L_jk ← max{L_{j-1,k}, L_{j,k-1}}
  return L
```

Laufzeit?

$\Theta(nm)$

Erklärung

Dies ist der komplette Algorithmus zur Berechnung der Längen der längsten gemeinsamen Untersequenzen. Wir sehen hier nun überhaupt nichts Neues mehr, denn er fasst nur noch einmal zusammen, was wir bereits ausführlich motiviert haben.

Der Algorithmus initialisiert die Tabelle L und berechnet die Einträge Zeile für Zeile. Für jeden Eintrag vergleicht er die letzten Zeichen der beiden Präfixe der Länge j bzw. k. Je nachdem, ob sie identisch sind oder nicht, wird die entsprechende, simple Berechnung ausgeführt, die wir soeben beschrieben haben.

Wir sehen mit einem Blick, dass die Laufzeit dieses Algorithmus $\Theta(nm)$ ist.

Dieses Vorgehen und dieses Resultat sind typisch für dynamische Programmierung: Man überlegt, wie man das Gesamtproblem so in gleichartige Unterprobleme zerlegen kann, dass sich die Lösungen in einer Tabelle berechnen lassen, wobei die Berechnungen auf bereits berechnete Werte zurückgreifen. Kaum hat man die Berechnungsvorschriften ausgearbeitet, ist der Algorithmus praktisch fertig.

Oft ist der fertige Algorithmus geradezu trivial einfach. Unser Algorithmus bestimmt die Längen der längsten gemeinsamen Untersequenzen zweier gegebener Sequenzen. Das ist auf den ersten Blick nicht ein unmittelbar einfaches Problem. Wer hätte gedacht, dass es sich mit einem extrem einfachen Algorithmus lösen lässt, der lediglich zwei geschachtelte Schleifen enthält, samt zweier Vergleiche, einer Inkrementierung und zweier Zuweisungen?

LCS in Aktion [Slide 14]

Animation

Sehen wir nun unseren `LCS()`-Algorithmus in Aktion auf den beiden Sequenzen $X =$`GATAACT` und $Y =$`AGGTCCTA`.

Zu Beginn wird die Tabelle L initialisiert, die eine Zeile und Spalte mehr hat, als die beiden Sequenzen lang sind, und deren erste Zeile und Spalte Null sind.

Die grünen Indizes in der Animation sind die Indizes der Tabelle. Diese geben definitionsgemäß die *Länge* der gezeigten Präfixe an, und sind daher um 1 größer als die Indizes der Zeichen innerhalb der Sequenzen.

Nun geht der Algorithmus die Tabelle zeilenweise durch. Bei jedem Eintrag vergleicht er die beiden Zeichen, für die dieser Eintrag steht, das heißt, die letzten Zeichen der beiden Sequenz-Präfixe, deren Längen durch die aktuellen Zeilen- und Spaltenindizes j und k gegeben sind.

Der erste Vergleich zwischen `G` und `A` fällt negativ aus, also wird $L_{1,1}$ auf das Maximum seines oberen und linken Nachbarn gesetzt, hier 0.

Beim nächsten Vergleich stimmen die beiden `G` überein. Dieses `G` beendet also eine längste gemeinsame Untersequenz, die um eins länger ist als die längste gemeinsame Untersequenz der beiden jeweils um dieses Zeichen verkürzten Präfixe. Deren Länge finden wir in unserem linken oberen Nachbarn, also in $L_{0,1} = 0$. Also setzen wir das aktuelle Feld $L_{1,2} = 0 + 1 = 1$.

Exakt das gleiche passiert im folgenden Vergleich für $L_{1,3}$.

Bei $L_{1,4}$ sind die beiden verglichenen Zeichen wieder verschieden, und wir setzen es auf das Maximum der linken und oberen Nachbarn. Hier ist es der linke mit Wert 1.

Diese geradezu langweilig einfache Prozedur wiederholt sich Zeile für Zeile. Wir sehen, dass sich die Tabelle mit Werten füllt, die nach rechts und nach unten monoton zunehmen. Dies ist keine Überraschung: Wenn wir die Präfixe der Sequenzen verlängern, dann können deren längste gemeinsame Untersequenzen allenfalls länger werden, aber niemals kürzer.

Ganz am Ende der Tabelle angekommen, finden wir die Länge der längsten gemeinsamen Untersequenzen von `GATAACT` und `AGGTCCTA` im letzten Feld: $L_{7,8} = 4$.

Rekonstruktion der Sequenz [Slide 15]

Video 4 beginnt hier.

Beginnend mit $j = n, k = m$:

- Falls $x_{j-1} = y_{k-1}$, gehört x_{j-1} zur Untersequenz; fahre bei $L_{j-1,k-1}$ fort;
- andernfalls fahre bei $\operatorname{argmax}\{L_{j,k-1}, L_{j-1,k}\}$ fort,

bis $L_{j,k} = 0$.

		A	G	G	T	C	C	T	A
		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
G	1	0	0	1	1	1	1	1	1	1
A	2	0	1	1	1	1	1	1	1	2
T	3	0	1	1	1	2	2	2	2	2
A	4	0	1	1	1	2	2	2	2	3
A	5	0	1	1	1	2	2	2	2	3
C	6	0	1	1	1	2	3	3	3	3
T	7	0	1	1	1	2	3	3	4	4

$$X = \overset{0}{G}\ \overset{1}{A}\ \overset{2}{T}\ \overset{3}{A}\ \overset{4}{A}\ \overset{5}{C}\ \overset{6}{T}$$

$$Y = \underset{0}{A}\ \underset{1}{G}\ \underset{2}{G}\ \underset{3}{T}\ \underset{4}{C}\ \underset{5}{C}\ \underset{6}{T}\ \underset{7}{A}$$

Andere längste gemeinsame Untersequenz: `ATCT`

Wie finden wir diese? Wie finden wir alle?

Erklärung

Wir haben bisher lediglich die *Länge* der längsten gemeinsamen Untersequenzen gefunden, aber nicht diese Untersequenzen selbst. Wie finden wir nun zumindest eine längste gemeinsame Untersequenz?

Die Idee ist, den Entstehungsprozess der Tabelle zurückzuverfolgen. Die entscheidende Information steckt in den Übergängen, bei denen sich die Länge erhöht. Diese Übergänge sind in dieser Abbildung fett nachgezeichnet.

Entlang dieser Übergänge muss es immer mindestens ein Feld j, k geben, für das die Zeichen in Zeile und Spalte identisch sind. Solche Zeichen gehören zu einer längsten gemeinsamen Untersequenz der Sequenz-Präfixe der Länge j bzw. k.

Wir beginnen also ganz am Ende, im Feld $L_{7,8}$. Hier stimmen das `T` und das `A` nicht überein. Also gehen wir zu einem linken oder oberen Nachbarn mit gleich hohem Wert, der existieren muss. Hier ist es eindeutig der linke Nachbar $L_{7,7}$. An dieser Stelle steht in beiden Sequenzen ein `T`, also gehört dieses `T` zu mindestens einer längsten gemeinsamen Untersequenz.

Weiter nach links können wir nicht gehen, denn $L_{7,6} < L_{7,7}$. Also haben wir keine Wahl, und dieses `T` ist folglich das letzte Zeichen *jeder* längsten gemeinsamen Untersequenz dieser beiden Sequenzen.

Nun schneiden wir beide Sequenzen vor diesem letzten Zeichen ab und suchen eine längste gemeinsame Untersequenz der beiden verbleibenden Präfixe. Mit anderen Worten, wir dekrementieren j und k und fahren auf dieselbe Weise bei $L_{6,6}$ fort.

Hier finden wir in beiden Sequenzen ein `C`. Dieses gehört also ebenfalls zu mindestens einer längsten gemeinsamen Untersequenz.

An dieser Stelle könnten wir wieder j und k dekrementieren und fortfahren. Dies ist genau das, was unsere Beschreibung links vorschlägt. Hier haben wir allerdings die Wahl, denn unser linker Nachbar $L_{6,5}$ hat ebenfalls den Wert 3, und seine beiden Zeichen stimmen wieder überein! Weitere Wahlmöglichkeiten haben wir jedoch nicht. Damit steht fest, dass alle längsten gemeinsamen Untersequenzen dieser beiden Sequenzen auf `CT` enden, und dass wir in unserer Sequenz Y zwischen y_5 und y_4 wählen können.

Ausgehend vom Feld unserer Wahl, hier $L_{6,5}$, dekrementieren wir wieder j und k und machen weiter.

Bei $L_{5,4}$ stimmen die beiden Zeichen x_4 =`A` und y_3 =`T` nicht überein, und die Werte

unserer Nachbarn zwingen uns nach oben.

Die nächste Übereinstimmung finden wir bei $L_{3,4}$. Dieses `T` ist eindeutig; wir können keinen Schritt nach links oder nach oben gehen, ohne dass sich der Wert des Tabellenfelds verkleinert. Also fahren wir bei $L_{2,3}$ fort.

Hier haben wir keine Übereinstimmung zwischen x_1 =`A` und y_2 =`G`, aber sowohl unser linker Nachbar $L_{2,2}$ als auch unser oberer Nachbar $L_{1,3}$ haben denselben Wert wie $L_{2,3}$. Wir haben also die Wahl, ob wir nach links oder nach oben gehen.

Hier gehen wir nach oben und finden bei $L_{1,3}$ übereinstimmend ein `G`. Dieses `G` ist also ebenfalls Teil einer längsten gemeinsamen Untersequenz. Alternativ können wir auch noch einen Schritt nach links gehen und y_1 statt y_2 verwenden.

Wären wir im vorigen Schritt von $L_{2,3}$ statt nach oben nach links gegangen, dann hätten wir keine Übereinstimmung zwischen x_1 =`A` und y_1 =`G` angetroffen. Hier in $L_{2,2}$ hätten wir aber wiederum die Wahl gehabt, ob wir nach links oder nach oben gehen. Wären wir nach oben gegangen, hätten wir wiederum das gemeinsame `G` angetroffen. Wären wir aber nach links gegangen, dann hätten wir bei $L_{2,1}$ ein übereinstimmendes `A` aufgefunden.

Damit hätten wir also eine *andere* längste gemeinsame Untersequenz gefunden, die mit `A` beginnt statt mit `G`. Indem wir alle Wahlmöglichkeiten berücksichtigen, die wir unterwegs angetroffen haben, können wir also feststellen, dass unsere beiden Sequenzen `GATAACT` und `AGGTCCTA` zwei verschiedene längste gemeinsame Untersequenzen aufweisen, nämlich `ATCT` und `GTCT`.

Algorithmus: Rekonstruktion einer Sequenz [Slide 16]

```
Algorithm reconstructLCS(X, Y, L):
  Require: as required and returned by Algorithm LCS.
  Ensure:  S is a longest common subsequence of X and Y.
  S ← empty sequence
  j ← n
  k ← m
  while L_jk > 0 do      // common characters remain
    if x_{j-1} = y_{k-1} then
      S ← x_{j-1}, S  // prepend common character to S
      j ← j − 1
      k ← k − 1
    else if L_{j-1,k} ≥ L_{j,k-1} then
      j ← j − 1
    else
      k ← k − 1
  return S
```

Übungen:

- Bestimmen Sie die asymptotische Laufzeit dieses Algorithmus.
- Schreiben Sie einen rekursiven Algorithmus zum Auffinden sämtlicher längster gemeinsamer Untersequenzen, und bestimmen Sie seine asymptotische Laufzeit.

Erklärung

Dieser Algorithmus zur Rekonstruktion einer längsten gemeinsamen Untersequenz ist wohl der einfachste denkbare: Bei jeder Übereinstimmung wird das übereinstimmende Zeichen verwendet und links darüber in L weitergemacht. Etwaige Wahlmöglichkeiten werden komplett ignoriert.

Möchten wir *alle* längsten gemeinsamen Untersequenzen finden, dann müssen wir *alle* Wahlmöglichkeiten verfolgen. Dies lässt sich recht einfach durch einen rekursiven Algorithmus bewerkstelligen.

4 Epilog

Dynamische Programmierung [Slide 17]

- gleichartige ***Unterprobleme***, die aufeinander aufbauen
- ***Optimalität der Unterprobleme***
- ***Überlappung*** der Unterprobleme
- Optimierung der Zielfunktion durch Ausfüllen einer ***Tabelle*** mit den Werten der Zielfunktion der Lösungen der Unterprobleme
- Rekonstruktion der Lösung anhand der ausgefüllten Tabelle sowie u.U. Hilfstabellen

Erklärung

Dynamische Programmierung ist von großer praktischer Bedeutung, weil sie einfache und effiziente Algorithmen für augenscheinlich komplexe Probleme hervorbringen kann.

Trotz der typischen Einfachheit des fertigen Algorithmus gibt es jedoch kein Standardrezept für seinen Entwurf.

Der entscheidende Baustein eines *Dynamic-Programming*-Algorithmus ist eine geeignete Formulierung auf Basis von Unterproblemen. Eine solche ist oft nicht offensichtlich.

Hat man sie jedoch gefunden, dann ist man meist schon so gut wie fertig, denn sie lässt sich typischerweise sehr direkt in einen Algorithmus umsetzen.

Die Unterprobleme formulieren wir auf Basis der Zielfunktion. Bei der Suche nach den längsten gemeinsamen Untersequenzen ist die Zielfunktion die Länge dieser Untersequenzen, die wir maximieren möchten. Also betrachten wir die *Maximierung der Länge* der gemeinsamen Untersequenzen als das zentrale Problem, und verschieben die Bestimmung der eigentlichen Untersequenzen auf später.

Auch dies ist typisch für dynamische Programmierung. Oft lässt sich die Lösung des ursprünglichen Problems aus der Tabelle rekonstruieren, wie es bei uns der Fall war. In anderen Fällen ist es hilfreich, während des Ausfüllens der Tabelle weitere Datenstrukturen zu befüllen, die eine Rekonstruktion der Lösung des ursprünglichen Problems ermöglichen.

In jedem Fall ist dynamische Programmierung höchst interessant und spannend, denn sie erfordert ein tiefes Verständnis des Problems und ein gesundes Maß an Kreativität, aber keine langwierige Routinearbeit. Und das Ergebnis ist meist kurz, knackig, elegant und effizient.

5 Literatur

Goodrich, Michael, Roberto Tamassia und Michael Goldwasser (Aug. 2014). *Data Structures and Algorithms in Java*. Wiley.