

Algorithmen und Datenstrukturen

Dynamische Programmierung

Prof. Justus Piater, Ph.D.

1. Juni 2022

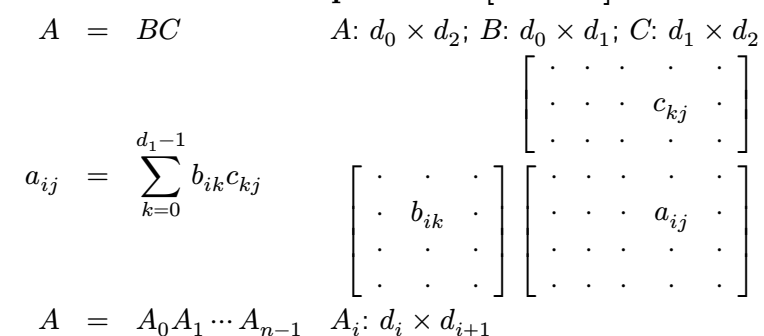
Dieser Kurs folgt in weiten Teilen dem sehr empfehlenswerten Lehrbuch
Data Structures and Algorithms in Java [Goodrich u. a. 2014].

Inhaltsverzeichnis

1	Matrix-Kettenmultiplikation	1
2	Paradigma	4
3	Längste gemeinsame Untersequenzen	8
4	Epilog	16
5	Literatur	16

1 Matrix-Kettenmultiplikation

Matrix-Kettenmultiplikation [Slide 1]

$$A = BC \quad A: d_0 \times d_2; B: d_0 \times d_1; C: d_1 \times d_2$$
$$a_{ij} = \sum_{k=0}^{d_1-1} b_{ik} c_{kj}$$
$$A = A_0 A_1 \cdots A_{n-1} \quad A_i: d_i \times d_{i+1}$$


Matrix-Multiplikation ist *assoziativ*.

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

$$10 \cdot 2 \cdot 50 + 50 \cdot 2 \cdot 20 = 3000$$
$$50 \cdot 10 \cdot 20 + 10 \cdot 2 \cdot 20 = 10400$$

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

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

Struktur des Problems [Slide 2]

- Das Problem besteht aus **Unterproblemen** $A_i A_{i+1} \dots A_j$; 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).

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

Beweis?

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

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

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 mit $N_{ij} = \min_{i \leq k < j} \{N_{ik} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$ [Slide 3]

$i = 2$			$N_{2,2}$	$N_{2,3}$	$N_{2,4}$		
					$N_{2+1,5}$		
					$N_{3+1,5}$		
					$N_{4+1,5}$		

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, \dots, d_n):`

Require: first dimensions d_0, \dots, d_{n-1} of matrices A_0, \dots, A_{n-1} ,
last dimension d_n of A_{n-1} .

Ensure: N is an $n \times n$ table with N_{ij} the number of multiplications
required to compute $A_i A_{i+1} \dots A_j$.

$N \leftarrow n \times n$ table with all-zero diagonal

for $b \leftarrow 1$ **to** $n - 1$ **do** // # matrix products in subchain

for $i \leftarrow 0$ **to** $n - b$ **do** // start of subchain

$j \leftarrow i + b$ // end of subchain

$N_{ij} \leftarrow \infty$

for $k \leftarrow i$ **to** $j - 1$ **do**

$N_{ij} \leftarrow \min\{N_{ij}, N_{ik} + N_{k+1,j} + d_i d_{k+1} d_{j+1}\}$

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

Algorithmisches Paradigma: *Dynamische Programmierung (Dynamic Programming)* [Slide 6]

Teile-und-Herrsche-Algorithmen teilen ein Problem in *disjunkte* Unterprobleme auf, die dann rekursiv *unabhängig voneinander* gelöst werden. Viele Probleme lassen sich nicht 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 Teilproblems auf Lösungen bereits gelöster Teilprobleme zurückgegriffen wird.

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. Dies macht die Programmplanung zu einem höchst dynamischen Prozess.

Diese Deutung des Begriffs *dynamische Programmierung* ist allerdings nur eine hilfreiche Anekdote; sein genauer Ursprung ist unbekannt.

Fibonacci: Rekursion (ineffizient) [Slide 7]

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

Algorithm `Fibonacci(k)`:

Require: Integer $k \geq 0$.

Ensure: Return the k -th Fibonacci number.

if $k \leq 1$ **then**

return k

return `Fibonacci(k - 2) + Fibonacci(k - 1)`

Schauen wir uns einmal die gute alte Fibonacci-Folge an, um dynamische Programmierung zu motivieren und zu illustrieren. Hier sehen wir ihre rekursive Definition und deren unmittelbare 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 von wiederholt berechneten Fibonacci-Zahlen.

Dynamische Programmierung vermeidet solche Mehrfachberechnungen überlappender Unterprobleme.

Fibonacci: Rekursion mit Memoisation (effizient) [Slide 8]

Algorithm `Fibonacci(k)`:

Require: integer $k \geq 0$.

Ensure: the k -th Fibonacci number.

```
if known[k]  $\neq$  0 then
```

```
    return known[k]
```

```
if  $k \leq 1$  then
```

```
    return k
```

```
known[k]  $\leftarrow$  Fibonacci( $k - 2$ ) + Fibonacci( $k - 1$ )
```

```
return known[k]
```

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

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 9]

Algorithm `Fibonacci(k)`:

Require: integer $k \geq 0$.

Ensure: the k -th Fibonacci number.

```
F[0]  $\leftarrow$  0
```

```
F[1]  $\leftarrow$  1
```

```
for  $i \leftarrow 2$  to  $k$  do
```

```
    F[i]  $\leftarrow$  F[i - 2] + F[i - 1]
```

```
return F[k]
```

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

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 $F[i - 1]$.

Hier sehen wir einige der charakteristischen Eigenschaften der dynamischen Programmierung in Aktion:

1. Das Problem der Berechnung von $F[k]$ lässt sich auf gleichartige *Unterprobleme* zurückführen, nämlich auf $F[k-2]$ und $F[k-1]$.
2. Die *Optimalität* dieser Unterprobleme besteht hier trivialerweise in der Korrektheit ihrer Berechnungsvorschrift, denn jedes Unterproblem $F[i]$ besitzt nur eine einzige Lösung.
3. Die Unterprobleme *überlappen* massiv, denn die Berechnung von $F[i-1]$ beinhaltet die komplette Berechnung von $F[i-2]$.
4. Zur effizienten Lösung des Problems befüllen wir die Tabelle F mit den Lösungen der Unterprobleme, iterativ jeweils aufbauend auf bereits vorhandenen Einträgen.

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.

3 Längste gemeinsame Untersequenzen

Längste gemeinsame Untersequenzen [Slide 10]

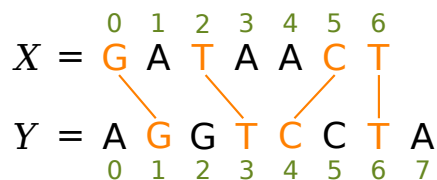
Eine **Untersequenz** einer Sequenz $X = x_0x_1 \cdots x_{n-1}$ ist eine Sequenz S der Form $x_{i_1}x_{i_2} \cdots x_{i_k}$ mit $i_j < i_{j+1}$.

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.

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 der 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 . Insgesamt 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 die ersten A 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 die 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*: Jede maximale gemeinsame Untersequenz der Länge $L_{j' \geq j, k' \geq k}$ muss eine maximale gemeinsame Untersequenz der Länge L_{jk} enthalten.

Beweis?

- *Überlappende Unterprobleme*: Dieselben Untersequenzen sind Teil verschiedener, längerer Untersequenzen.

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 offensichtliche 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 ein weiteres Zeichen. Hilft uns unsere Kenntnis der längsten gemeinsamen Untersequenzen der alten Versionen von X und Y bei der Bestimmung der längsten gemeinsamen Untersequenzen der verlängerten Versionen?

Ja, das tut sie in der Tat. Wir müssen nicht von vorn beginnen, sondern müssen lediglich das neu angehängte Zeichen von X mit dem neu angehängten Zeichen von Y vergleichen. Sind sie identisch, dann können wir die alte längste gemeinsame Untersequenz um dieses Zeichen verlängern.

Schreiben wir nun diese Erkenntnis formal auf, im Sinne der drei charakteristischen Eigenschaften der dynamischen Programmierung.

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 .

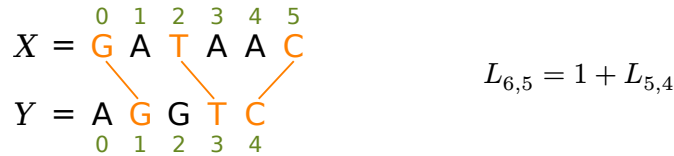
2. besitzt diese Formulierung die Eigenschaft der *Optimalität der Unterprobleme*, denn die längsten gemeinsamen Untersequenzen von Präfixen von X und Y müssen Teil der längsten gemeinsamen Untersequenzen längerer Präfixe von X und Y sein. Mit anderen Worten, das Anhängen eines weiteren Zeichens an X kann die längsten gemeinsamen Untersequenzen mit Y nicht um mehr als dieses eine Zeichen verlängern. Denn ist dieses angehängte Zeichen tatsächlich Teil einer längsten gemeinsamen Untersequenz, dann wird sich diese längste gemeinsame Untersequenz um maximal dieses eine Zeichen verkürzen, wenn ich es wieder entferne.
3. überlappen diese Unterprobleme, denn die längsten gemeinsamen Untersequenzen einer sehr kurzen Version von X mit Y sind Teil der längsten gemeinsamen Untersequenzen einer verlängerten Version von X mit Y , diese sind wiederum Teil der Lösung mit einer nochmals verlängerten Version von X , und so weiter.

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

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 $L_{jk} = 1 + L_{j-1,k-1}$.



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 $L_{jk} = \max \{L_{j-1,k}, L_{j,k-1}\}$.



Die Berücksichtigung des nächsten Zeichens in X bzw. Y kann also nicht die längste Untersequenz verlängern, die X bzw. Y ohne dieses Zeichen gemeinsam haben.

Nun müssen wir nur noch 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 $y_3 = T$. 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 $X = x_0 \cdots x_{n-1}$ and $Y = y_0 \cdots y_{m-1}$.

Ensure: L is a table with L_{jk} the length of

an LCS of $x_0 \cdots x_{j-1}$ and $y_0 \cdots y_{k-1}$.

$L \leftarrow (n+1) \times (m+1)$ table with all-zero 1st column and row

for $j \leftarrow 1$ **to** n **do**

for $k \leftarrow 1$ **to** m **do**

if $x_{j-1} = y_{k-1}$ **then**

$L_{jk} \leftarrow L_{j-1,k-1} + 1$

else

$L_{jk} \leftarrow \max\{L_{j-1,k}, L_{j,k-1}\}$

return L

Laufzeit?

$\Theta(nm)$

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 ge-

schachtelte Schleifen enthält, samt zweier Vergleiche, einer Inkrementierung und zweier Zuweisungen?

LCS in Aktion [Slide 14]

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

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]

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	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
		0	1	2	3	4	5	6		
X =		G	A	T	A	A	C	T		
Y =		A	G	G	T	C	C	T	A	
		0	1	2	3	4	5	6	7	

Andere längste gemeinsame Untersequenz: ATCT

Wie finden wir diese? Wie finden wir alle?

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 .

$S \leftarrow$ empty sequence

$j \leftarrow n$

$k \leftarrow m$

while $L_{jk} > 0$ **do** // common characters remain

if $x_{j-1} = y_{k-1}$ **then**

$S \leftarrow x_{j-1}, S$ // prepend common character to S

$j \leftarrow j - 1$

$k \leftarrow k - 1$

else if $L_{j-1,k} \geq L_{j,k-1}$ **then**

$j \leftarrow j - 1$

else

$k \leftarrow k - 1$

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.

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

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.