

13.1 Grundprinzip und einfaches Beispiel

13.2 Rucksackproblem

13.3 Matrix-Multiplikation

13.4 Approximative Zeichenkettensuche

13.5 Suchmuster in Texten

13.6 Amortisierte Analyse



Innensicht

Algorithmenentwurf und Algorithmen (incl. Aufwände)

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

Prozesse

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

Außensicht

Skalierbarkeit und Persistenz

- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Aufwände
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

Autonome Dienste

- Enge/lose Kopplung
- Protokolle / Automaten
- Verteilung



Gegensatz

- Teile und Herrsche
 - Top-down-Vorgehensweise
 - Zerlegung in Teilprobleme
- Dynamisches Programmieren
 - *Bottom-up*-Vorgehensweise
 - Konstruktion der Lösung zunächst für Teilprobleme
 - Häufige Variante: Vorbereitung von Teilen der Daten vor Inangriffnahme der eigentlichen Aufgabe (Vorausberechnung).

Rekursive Lösungsstruktur

- Aufteilung in *abhängige* Teilprobleme
- Berechnen und Zwischenspeichern wiederkehrender Teilergebnisse
- Bestimmung des Gesamtergebnisses unter Verwendung der Teilergebnisse



Effizient

- Wenn Teillösungen überlappend sind

Voraussetzung: Bellmannsches Optimalitätsprinzip

- Optimale Lösung eines Problems der Größe n ist im Innern aus optimalen Teillösungen kleinerer Größe zusammengesetzt.

Häufig verwendet bei Optimierungsproblemen

- Interpretation der Teilergebnisse als optimale Lösung eines "kleineren" Teilproblems

Vorgehensweise

- Charakterisiere Lösungsraum und Struktur einer optimalen Lösung
- Definiere rekursiv, wie sich eine optimale Lösung aus kleineren optimalen Lösungen zusammensetzt
- Konzipiere Algorithmus in einer bottom-up Weise
 - Tabellarisches Finden optimaler Teillösungen



Schöning, Algorithmen –
Kurz gefasst, Spektrum, 1997



zur Erinnerung aus Info I

Definition

$$F_0 = 0,$$

$$F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2)$$

Nach Teile-und-herrsche-Prinzip mit Rekursion

```
static int fib1(int n) {  
    if (n == 0) return 0;  
    else if (n == 1) return 1;  
    else return (fib1(n - 1) + fib1(n - 2));  
}
```



Beobachtung

- Berechnung wiederholt bestimmte Schritte

$$\text{fib}(2) = \text{fib}(1) + \text{fib}(0)$$

$$\text{fib}(3) = \text{fib}(2) + \text{fib}(1)$$

$$\text{fib}(4) = \text{fib}(3) + \text{fib}(2) \quad \dots$$

Aufwandsabschätzung

- $T(n)$ = Anzahl Additionen

$$T(0) = 0$$

$$T(1) = 0$$

$$T(n) = 1 + T(n-1) + T(n-2) \text{ für } n \geq 2$$

⇒ Rekursive Lösung ist teuer!

Es geht billiger mit dynamischer Programmierung.



Beobachtung

- $\text{fib}(n-2)$ wird im folgenden Rekursionsschritt zu $\text{fib}(n-1)$, daher unnötige Wiederholung der Berechnung.
- Abhilfe: Variablen fi1 und fi2 mit Fi-1 bzw. Fi-2 .

Programmierung

```
static int fib2(int n) {  
    if (n == 0) return 0;  
    else if (n == 1) return 1;  
    else {  
        int fi1 = 1; int fi2 = 0;  
        for (int i = 2; i <= n; i++)  
        { //berechnen aus bestehenden Teilergebnissen  
            int m = fi1 + fi2;  
            fi2 = fi1; fi1 = m; //merken der Teilergebnisse  
        }  
        return fi1;  
    }  
}
```

1. Durchlauf:

- $\text{fi2} = \text{fib}(0)$
- $\text{fi1} = \text{fib}(1)$

2. Durchlauf:

- $\text{fi2} = \text{fib}(1)$
- $\text{fi1} = \text{fib}(2)$
- $m = \text{fib}(3)$

...

Aufwand: Anzahl der Additionen

- $T(n) = n-1 \ll 1 + T(n-1) + T(n-2)$



Beispiel Schatzsucher

- Schatzsucher findet Schatz bestehend aus Goldklumpen.
- Jeder Goldklumpen hat
 - ein bestimmtes Gewicht und
 - einen bestimmten Wert.
- Schatzsucher hat einen Rucksack
 - Kapazität durch ein maximales Gewicht begrenzt



Die Aufgabe: befülle den Rucksack mit Goldklumpen

- ohne die Gewichtsbeschränkung zu verletzen
- bei Maximierung des Wertes der Goldklumpen.



Gegeben

- Kapazität $c \in \mathbb{IN}$
- Menge O mit $n \in \mathbb{IN}$ Objekten o_1, o_2, \dots, o_n
- Gewichtsfunktion $g: O \rightarrow \mathbb{IN}$
- Bewertungsfunktion $w: O \rightarrow \mathbb{IN}$

Gesucht ist $O' \subseteq O$ mit

$$\sum_{j \in O'} g(j) \leq c \text{ und } \sum_{j \in O'} w(j) \text{ maximal}$$

$$\text{Dabei gelte } \sum_{j \in O} g(j) > c$$

Anmerkung

- Eigtl. 0-1 Rucksack-Problem, da nur ganze Objekte betrachtet werden dürfen (ansonsten Bruchteil-Rucksack-Problem (Fractional Knapsack)).



Aber: Greedy funktioniert beim 0-1 Rucksack-Problem nicht!

Beispiel: $O = \{o_1, o_2, o_3\}$, Kapazität $c = 5$

- Gewichte: $g(o_1) = 1$, $g(o_2) = 2$, $g(o_3) = 3$
- Werte: $w(o_1) = 6$, $w(o_2) = 10$, $w(o_3) = 12$

Greedy

- Nimm Objekt mit größtem relativen Wert bis Rucksack voll!

Relative Werte ($r(o) = w(o)/g(o)$)

- $r(o_1) = 6$, $r(o_2) = 5$, $r(o_3) = 4$

Ergebnis: $O' = \{o_1, o_2\}$, mit

- $\sum_{j \in O'} g(j) = 3 < c = 5$

- $\sum_{j \in O'} w(j) = 16$

- sicher nicht maximal! Besser ist $O'' = \{o_3, o_2\}$ mit
 $\sum_{j \in O''} g(j) = 5 \leq c = 5$ $\sum_{j \in O''} w(j) = 22$

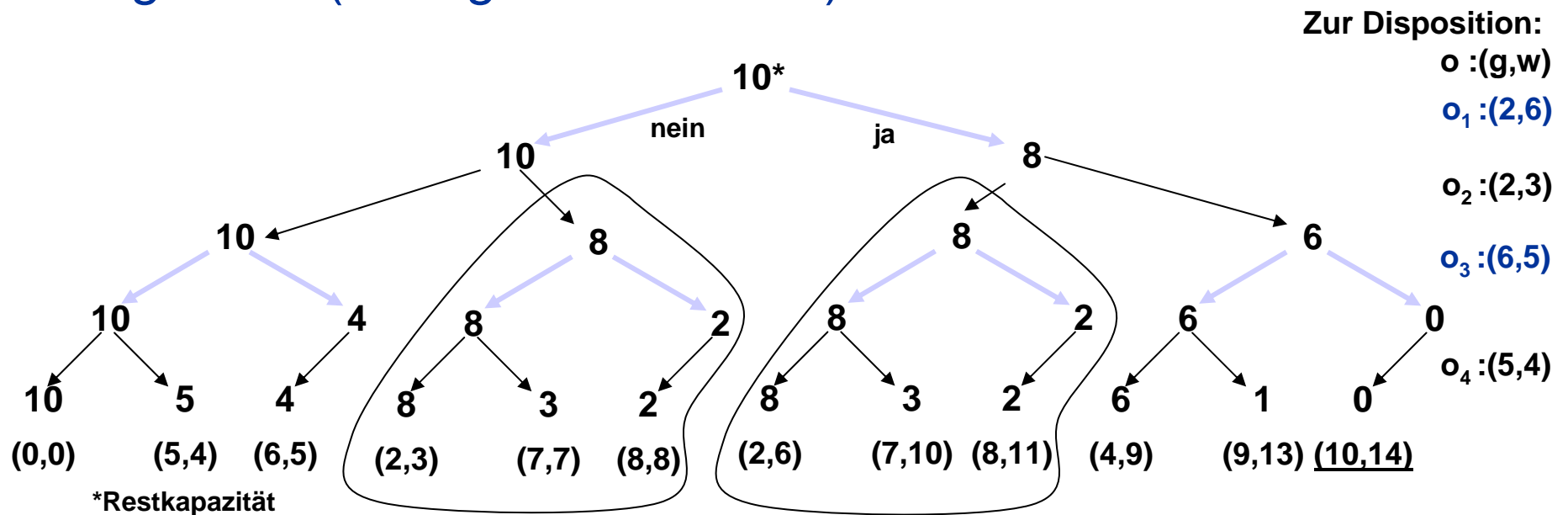


Funktioniert, ist aber aufwändig!

Beispiel: $O = \{ o_1, o_2, o_3, o_4 \}$, $c = 10$

- Gewichte: $g(o_1) = 2$, $g(o_2) = 2$, $g(o_3) = 6$, $g(o_4) = 5$
- Werte: $w(o_1) = 6$, $w(o_2) = 3$, $w(o_3) = 5$, $w(o_4) = 4$

Lösungsraum (Konfigurationsbaum)



Teilbäume gleich → Optimierungspotenzial
 → Dynamische Programmierung



```
public int btKnapsack(int i, int restkapazität ) {  
    // Aufruf: btKnapsack(1,c)  
    // Rückgabewert: optimaler Wert mit/ohne Objekt  
    Objekt o=objekte[i];  
    if ( i==objekte.length ) // Ende der Rekursion  
        if (o.gewicht() > restkapazität) // o bleibt draußen  
            return 0;  
        else return o.wert(); // o rein, Steigerung des Werts  
    else if (o.gewicht() > restkapazität)  
        return btKnapsack(i+1, restkapazität)  
    else return max(btKnapsack(i+1,restkapazität),  
        btKnapsack(i+1,restkapazität-o.gewicht())+o.wert());  
        // Wenn es noch hinein passt, werden dennoch  
        // beide Möglichkeiten betrachtet  
}
```

Anmerkung

- Optimierungspotenzial (Ausnutzen wiederkehrender Berechnungen) wird hier nicht genutzt.



Objekt einpacken oder nicht

- Objekt i wird nicht eingepackt
 - `btKnapsack(i+1, restkapazität)`
- Objekt i wird eingepackt
 - `max(btKnapsack(i+1, restkapazität),
btKnapsack(i+1, restkapazität -
o.gewicht()) + o.wert())`

Optimalitätsprinzip erfüllt

- Falls Rucksack der Größe G optimal mit einer Auswahl der Objekte $\{o_1, \dots, o_n\}$ gepackt ist, gilt dass ein $(G - g(o_i))$ -großer Rucksack optimal mit einem Teil der Objekte $\{o_1, \dots, o_n\} - \{o_i\}$ gepackt ist.



Idee: Rückwärts die Resultate der Aufrufe

btKnapsack(i, r) berechnen

- Verwendung eines zweidimensionalen Feldes f
 - $f(i, r) = \text{btKnapsack}(i, r)$

Beispiel: $O = \{ o_1, o_2, o_3, o_4, o_5 \}, c = 10$

- Gewichte: $g(o_1) = 2, g(o_2) = 2, g(o_3) = 6, g(o_4) = 5, g(o_5) = 4$
- Werte: $w(o_1) = 6, w(o_2) = 3, w(o_3) = 5, w(o_4) = 4, w(o_5) = 6$

i \ r	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11

- Beispiel*: $f[4,9] = \text{btKnapsack}(4,9) = 10$ (d.h. o_4 und o_5 haben bei Restkapazität 9 noch Platz: Wert der Füllung = 10)

*Anstelle der Java-üblichen Schreibweise $f[4][9]$ wird im Folgenden $f[4,9]$ verwendet.



Beachte zentrale Anweisungen in `btKnapsack(i, r)`:

```
... else return max(btKnapsack(i+1, restkapazität),  
    btKnapsack(i+1, restkapazität-o.gewicht())+o.wert()) ...
```

Bsp.:

	0	1	2	3	4	5	6	7	8	9	10
5	0	0	0	0	6	6	6	6	6	6	6
4	0	0	0	0	6	6	6	6	6	10	10
3	0	0	0	0	6	6	6	6	6	10	11
2	0	0	3	3	6	6	9	9	9	10	11

$$f[3,8] = \max(f[4,8], f[4,2]+5) = \max(6, 0+5) = 6$$

Idee für dynamische Programmierung verfeinert:

1. Berechnung der Werte für $i=n$ und alle Restkapazitäten von 0 bis c .
2. Dann von $i=n-1$ bis 2 Berechnung der Feldeinträge für jeweils alle Kapazitäten unter Rückgriff auf die bereits berechneten Werte der Zeile $i+1$.
3. Gesamtergebnis $f[1,c]$ dann durch Einzelberechnung
 $(f[1,10] = \max(f[2,10], f[2,8]+6) = \max(11, 9+6) = 15).$



```
public int dpKnapsack(int n, int c ) {  
    int[][] f = new int[2,n][0,c];  
    // Initialisierung des Felds für i=n  
    for (int r = 0; r<=c; r = r+1)  
        if ( objekte[n].gewicht() > r ) f[n,r] = 0;  
        else f[n,r] = objekte[n].wert();  
    // Berechnung des Felds für i = n-1,...,2  
    for (int i = n-1; i >=2; i = i-1)  
        for (int r = 0; r<=c; r = r+1)  
            if ( objekte[i].gewicht() > r ) f[i,r] = f[i+1,r];  
            else f[i,r] = max( f[i+1,r], f[i+1, r-  
                objekte[i].gewicht()]  
                +objekte[i].wert() );  
    // Berechne f[1,c] (den maximal erreichbaren Wert)  
    if ( objekte[1].gewicht() > c ) return f[2,c];  
    else return max( f[2,c], f[2, c-objekte[1].gewicht()] +  
        objekte[1].wert() );  
}
```



Relevante Parameter

- Tabellengröße
- Aufwand pro Tabelleneintrag
- $O(nG) * O(1) = O(nG)$

Aber

- 0-1 Rucksack-Problem ist NP-vollständig!
- Beachten der Bitkomplexität
 - $G = 2^k \rightarrow O(n2^k)$

Verfahren nicht anwendbar, wenn

- Größen oder Werte beispielsweise reelle Zahlen sind, also keine ganzen Zahlen



Zur Erinnerung

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{11} \\ b_{21} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} \\ a_{21}b_{11} + a_{22}b_{21} \end{pmatrix}$$

Matrix A Matrix B Matrix C
(2x2) (2x1) (2x1)

- Matrizenmultiplikation durch elementweise Skalar-Multiplikation und Addition
- In diesem Beispiel: 4 Skalar-Multiplikationen (2x2x1)
- Aber was geschieht bei einer verketteten Multiplikation?

Betrachte die Multiplikation von n Matrizen M_1, M_2, \dots, M_n

$$P = M_1 M_2 \dots M_n$$

Matrizen-Bedingung erfüllt:
Anzahl Spalten der Matrix M_i =
Anzahl Zeilen der Matrix M_{i+1}



Klammerung bestimmt Reihenfolge der Multiplikation

- Wie sind $M_1 M_2 \dots M_n$ geklammert?
- Vollständig geklammert, falls Matrix einzeln oder Produkt von zwei Matrizen
 - Beispiel: $(M_1((M_2 M_3) M_4))$
- Matrizenmultiplikation ist assoziativ, d.h. alle Klammerungen führen zum selben Ergebnis.

Komplexität

- Abhängig von der Anzahl an Skalar-Multiplikationen
- → hängt von der Klammerung ab



Beispiel: 6 Matrizen

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} (d_{11} \quad d_{12}) \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

Matrix A B C D E F
 (4x2) (2x3) (3x1) (1x2) (2x2) (2x3)

Frage: wie hoch ist die Gesamtzahl der benötigten Skalar-Multiplikationen?

- Erste Idee eines Multiplikations-Algorithmus:

Multipliziere zunächst Matrix A mit B

24 Multiplikationen (4x2x3)

Multipliziere das Ergebnis mit Matrix C

+12 Multiplikationen (4x3x1)

Multipliziere das Ergebnis mit Matrix D

+8 Multiplikationen (4 x1x2)

... mit Matrix F



Sedgewick; Algorithmen,
Addison-Wesley, 1995



- Insgesamt 84 Skalar-Multiplikationen notwendig, bei dieser Art des Vorgehens: „von links nach rechts Multiplizieren“
 - Klammerung: (((((AB)C)D)E)F)

Beobachtung

- Wenn wir stattdessen von „rechts nach links Multiplizieren“
 - Klammerung: (A(B(C(D(EF)))))

⇒ insgesamt nur 69 Skalar-Multiplikationen!

- Offenbar hängt die Anzahl der Skalar-Multiplikationen von der Reihenfolge der Matrix-Multiplikationen ab.
- In diesem Beispiel ist der Unterschied relativ gering (15 Skalar-Multiplikationen).
- Komplexeres Beispiel
 - Betrachte B, C, F nicht mit Dimension 3 sondern 300
 - rechts-links ⇒ 6024 Skalar-Multiplikationen
 - links-rechts ⇒ 274.200 Skalar-Multiplikationen



Faktor 45!!!



Ziel

- Bestimme die Reihenfolge der Matrix-Multiplikationen, die die wenigsten Skalar-Multiplikationen benötigt.

Problemstellung

- Gegeben: Kette von n Matrizen $M_1 M_2 \dots M_n$ wobei Matrix M_i die Dimension $p_{i-1} \times p_i$ hat
- Gesucht: Vollständig geklammertes Produkt, das die Anzahl der Skalar-Multiplikationen minimiert

Bemerkung

- Nicht die Multiplikation selbst ist von Interesse, sondern das Finden einer Reihenfolge mit minimalem Aufwand



Einfaches Ausprobieren aller möglichen Reihenfolgen keine gute Idee!

Bestimmung der Anzahl möglicher Klammerungen

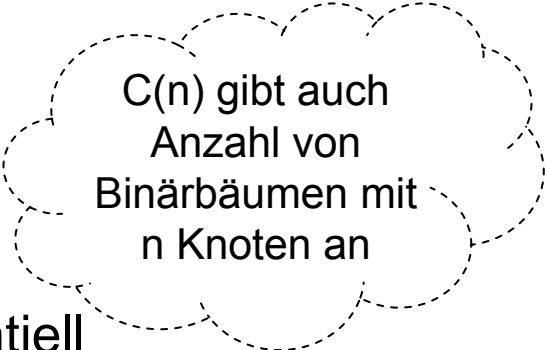
- Sei $K(n)$ die Anzahl der verschiedenen Möglichkeiten
- $n-1$ Möglichkeiten für die äußeren Klammern
- Danach $M_1 \dots M_j$ und $M_{j+1} \dots M_n$ betrachtet werden müssen, wobei jeweils $K(j)$ und $K(n-j)$ Klammerungen möglich sind
- Rekursion

$$K(n) = \sum_{j=1}^{n-1} K(j) \cdot K(n-j), \quad K(1) = 1$$

Lösung der Rekursionsgleichung: Catalansche Zahlen

- $K(n) = C(n-1)$

- $$C(n) = \binom{2n}{n} \cdot \frac{1}{n+1} = \Omega\left(\frac{4^n}{n^{3/2}}\right)$$



$C(n)$ gibt auch Anzahl von Binärbäumen mit n Knoten an

Komplexität

- vom naiven Durchprobieren ist also exponentiell



Anzahl Klammerungen

- Im Beispiel mit 6 Variablen (Matrizen) gibt es 132 Möglichkeiten zur Klammerung
- Bei 10 Matrizen schon 16796 Möglichkeiten.

Bemerkung: Wir betrachten das Standard-Verfahren zur Multiplikation von Matrizen ($O(n^3)$). Es gibt sicherlich Methoden, die 2 Matrizen effizienter multiplizieren, z.B. der Algorithmus von Strassen mit ($O(n^{2,8})$).



Notation

- Sei $M_{i..j}$ diejenige Matrix die aus der Multiplikation der Matrizen $M_i M_{i+1} \dots M_j$ entsteht.

Vorgehen

- Es sei $i < j$, Problem ist also nicht trivial lösbar
- Teilen des Produkts an einer Stelle $i \leq k < j$
 - Kosten: Multiplizieren der beiden Matrizen $M_{i..k}$ und $M_{k+1..j}$ sowie multiplizieren dieser Matrizen um das Ergebnis $M_{i..j}$ zu erlangen.
- Annahme
 - Optimale Klammerung teilt Produkt zwischen M_k und M_{k+1}
 - Dann muss Klammerung im Präfix optimal sein
 - Falls nicht, müsste es eine billigere Lösung geben, um $M_i M_{i+1} \dots M_k$ zu klammern. Einsetzen dieser Klammerung in die optimale Klammerung $M_i M_{i+1} \dots M_j$ würde eine weitere Klammerung hierfür produzieren mit geringeren Kosten. Widerspruch!
 - Ähnlich für anderen Teil der Klammerung $M_{k+1} \dots M_j$

Optimalitätsprinzip erfüllt



Teilproblem

- Finde die Klammerung mit minimalen Kosten für $M_i M_{i+1} \dots M_j$ für $1 \leq i \leq j \leq n$
- Sei $m[i, j]$ die minimale Anzahl an Skalar-Multiplikationen, um $M_{i..j}$ zu berechnen.

Gesamtkosten

- Die geringsten Kosten, um $M_{1..n}$ zu berechnen, sind dann $m[1, n]$

Rekursive Definition von $m[i, j]$

$$m[i, j] = \begin{cases} 0 & \text{falls } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{falls } i < j \end{cases}$$

Anmerkung

- Algorithmus zur darauf basierenden rekursiven Lösung hat exponentiellen Aufwand.



Beobachtung

- Problem hat wenige Teilprobleme
 - Ein Problem für jede Wahl von i und j
- Teilprobleme überlappen in mehreren Zweigen des Rekursionsbaums

Vorgehensweise

- Nutzen einer Tabelle zum Zwischenspeichern von Ergebnissen
- Bottom-up Ansatz



Löse (und speichere) Teilprobleme!

- Berechne Anzahl der notwendigen Skalar-Multiplikationen der Teilprodukte
 $M_1M_2, M_2M_3, \dots, M_{n-1}M_n$
- Berechne günstigste Anzahl der Triple-Produkte aus errechneten Teilprodukten

- Beispiel:

$$M_1M_2M_3\dots$$

- Schritt 1: $k = (M_1M_2)M_3$ k =Anzahl der Skalar-Multiplikationen bei $(M_1M_2)M_3$
 \Rightarrow Kosten für M_1M_2 + Kosten dazu M_3 zu multiplizieren
 $l = M_1(M_2M_3)$ l =Anzahl der Skalar-Multiplikationen bei $M_1(M_2M_3)$
 \Rightarrow Kosten für M_2M_3 + Kosten dazu M_1 zu multiplizieren
- Schritt 2: Zur Berechnung der günstigsten Skalar-Multiplikation von $M_1M_2M_3$ wähle k , wenn $k < l$, sonst l .
Speichere daraus entstehende Kosten $(M_1M_2M_3)$ ab.



- Verfahre analog mit allen anderen Tripeln
- Berechne nun die günstigste Möglichkeit Quadrupel zu multiplizieren
 - Verwende die gerade bestimmten günstigsten Tripel
 - Addiere und überprüfe Kosten für zusätzliche Multiplikation an diese Tripel
- Mit Quintupel weitermachen
- ...

Auf diese Art kann für die komplette Multiplikations-Kette die günstigste Reihenfolge bestimmt werden.

- Gegeben sei eine Multiplikationskette $M_i M_{i+1} \dots M_{i+j}$
- Bestimme für jedes k zwischen i und $i+j$ mit $M_i M_{i+1} \dots M_k$ und $M_{k+1} \dots M_{i+j}$ die Kosten
- Addiere Kosten, die bei Multiplikation dieser Teilergebnisse entstehen
- Speichere neue Kosten jeweils in einer Tabelle m (und entnehme sie von da)



Für $M_i M_{i+1} \dots M_k$ und $M_{k+1} \dots M_{i+j}$

- $m[i, r]$ speichere die minimalen Kosten, die bei der Berechnung von $M_i M_{i+1} \dots M_r$ entstehen, hier $m[i, k]$ und $m[k+1, i+j]$
 - r sei ein Array, das die Dimensionen der Matrizen aufnimmt, Matrix M_i hat die Dimension $r_i \times r_{i+1}$, gespeichert in $r[i]$ und $r[i+1]$
 - Kosten für die abschließende Multiplikation:
 $M_i M_{i+1} \dots M_k$ hat die Dimension $r_i \times r_{k+1}$ und $M_{k+1} \dots M_{i+j}$ die Dimension $r_{k+1} \times r_{i+j+1}$. \Rightarrow zusätzliche Kosten $r_i r_k r_{i+j+1}$.
 - Auf diese Weise $m[i, i+j]$ berechnen
-
- Im Array $best[i, j]$ merken wir uns das k , das die beste „Zerlegung“ in $M_i M_{i+1} \dots M_k$ und $M_{k+1} \dots M_{i+j}$ ergibt.



```
for (i=1;i<=n;i++)  
    for (j=i+1;j<=n;j++) m[i,j]=maxint;  
    // Initialisiere Kosten-Array: auf "maximal" setzen  
for (i=1;i<=n;i++) m[i,i]=0;  
    // Die Kosten von  $M_i$  nach  $M_i$  sind gleich 0  
for (j=1;j<=n-1;j++)  
    for (i=1;i<=n-j;i++) // laufe über alle Produkte  
        for (k=i+1;k<=i+j;k++) { // prüfe mögliche "Zerlegungen"  
            t=m[i,k]+m[k+1,i+j]+r[i]*r[k]*r[i+j+1];  
            if (t<m[i,i+j]) { // Wenn die neuen Kosten  
                // geringer sind..  
                m[i,i+j]=t; // Abspeichern und  
                best[i,i+j]=k; // neue "Zerlegung" merken  
            }  
        }  
    }
```

 $O(n^2)$ $O(n)$ $O(n^3)$

Mit dynamischer Programmierung wird das Problem der Multiplikation mehrerer Matrizen in $O(n^3)$ Schritten gelöst.



Sedgewick; Algorithmen,
Addison-Wesley, 1995



$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \\ a_{41} & a_{42} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} (d_{11} \quad d_{12}) \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

A (4x2) B (2x3) C (3x1) D (1x2) E (2x2) F (2x3)

	B	C	D	E	F
A	24 AB	14 A(BC)	22 (ABC)D	26 (ABC)(DE)	36 (ABC)(DEF)
B		6 BC	10 (BC)D	14 C(DE)	22 C(DEF)
C			6 CD	10 C(DE)	19 C(DEF)
D				4 DE	10 (DE)F
E					12 EF

(ABC)(DEF) bedeutet: multipliziere zunächst ABC optimal miteinander und das Ergebnis mit dem Ergebnis der optimalen Multiplikation von DEF



Sedgewick; Algorithmen,
Addison-Wesley, 1995



Vergleich von Symbolsequenzen

Internet-Suchmaschinen: Auffinden von Textstücken:

Bundeskanzler

- z.B. in Dokumenten („In welchen Dokumenten kommt **X** vor“).

... im Bundestag sagte der **Bundeskanzler**, er ...

- ggf. auch unter Berücksichtigung von Schreibfehlern:

... sagte der **Bundeskanler**, er ...



Einheitliche Betrachtung:

Anzahl der Editierschritte, um A in B zu überführen.

getippt: pribic gemeint private oder public ?

Aufgabe:

- Gegeben: Symbolfolgen $A = a_1, a_2, \dots, a_n$ und $B = b_1, b_2, \dots, b_m$
- Gesucht: Minimale „Editierdistanz“

Beispiel: $A = \text{pribic}$ $B = \text{public}$

- 1. Schritt: ersetze r durch u pribic \Rightarrow puibic
- 2. Schritt: lasse das erste i in A weg puibic \Rightarrow pubic
- 3. Schritt: füge ein l nach dem b ein pubic \Rightarrow public
- In weniger als 3 Schritten geht es nicht, in mehr als 3 aber wohl.



Editierdistanz

- Zeichenketten können über Ersetzungsregeln ineinander überführt werden
- Pro Regel wird an einer Stelle ein Zeichen verändert, in der Form $\alpha \rightarrow \beta$: ändere Zeichen α in Zeichen β um.
- Löschen $\alpha \rightarrow \varepsilon$: lösche Zeichen α
- Einfügen $\varepsilon \rightarrow \alpha$: füge Zeichen α hinzu
- Ändern $\alpha \rightarrow \beta$: ändere Zeichen α in Zeichen β

Beispiel
auto: $u \rightarrow \varepsilon$
ato: $t \rightarrow d$
ado: $o \rightarrow \varepsilon$
ad: $\varepsilon \rightarrow r$
 \Rightarrow rad

Jeder Editierschritt kostet!

- Pro Operation $\alpha \rightarrow \beta$ entstehen $c(\alpha \rightarrow \beta) > 0$ Kosten. c heißt Kostenfunktion.

Die **Editierdistanz** $C(A,B)$ bezeichnet die minimalen Kosten, die bei Überführung von Zeichenkette A in Zeichenkette B entstehen.



- Dreiecksungleichung sollte erfüllt sein: $c(\alpha \rightarrow \gamma) \leq c(\alpha \rightarrow \beta) + c(\beta \rightarrow \gamma)$
- Einheitskosten-Modell

Pro Löschen-, Einfügen-, Ändern-Operation jeweils 1 Kosteneinheit

$$1 = c(\alpha \rightarrow \varepsilon) = c(\varepsilon \rightarrow \alpha) = c(\alpha \rightarrow \beta) \text{ und } c(\alpha \rightarrow \alpha) = 0$$

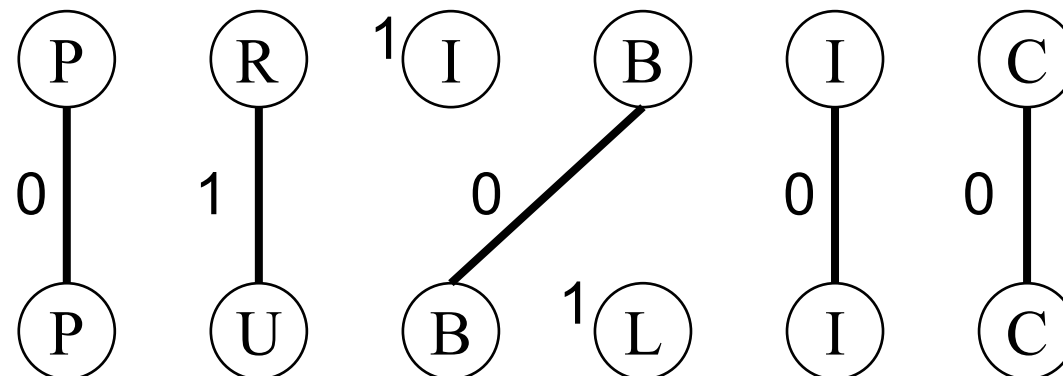


Problem: Wie kann man die Editierdistanz $C(A,B)$ für zwei gegebene Zeichenketten A und B effizient berechnen?

- Ändere jedes Zeichen höchstens einmal (Dreiecksungleichung)
- Führe alle Änderungen gleichzeitig aus

Spur

- Schreibe beide Zeichenketten untereinander
- Verbinde gleiche Zeichen mit einer Kante, Beschriftung 0
- Verbinde geänderte Zeichen mit einer Kante, Beschriftung 1
- Bei hinzugefügten Zeichen: schreibe 1 links neben das neue Zeichen
- Bei gelöschten Zeichen: schreibe 1 links neben das gelöschte Zeichen

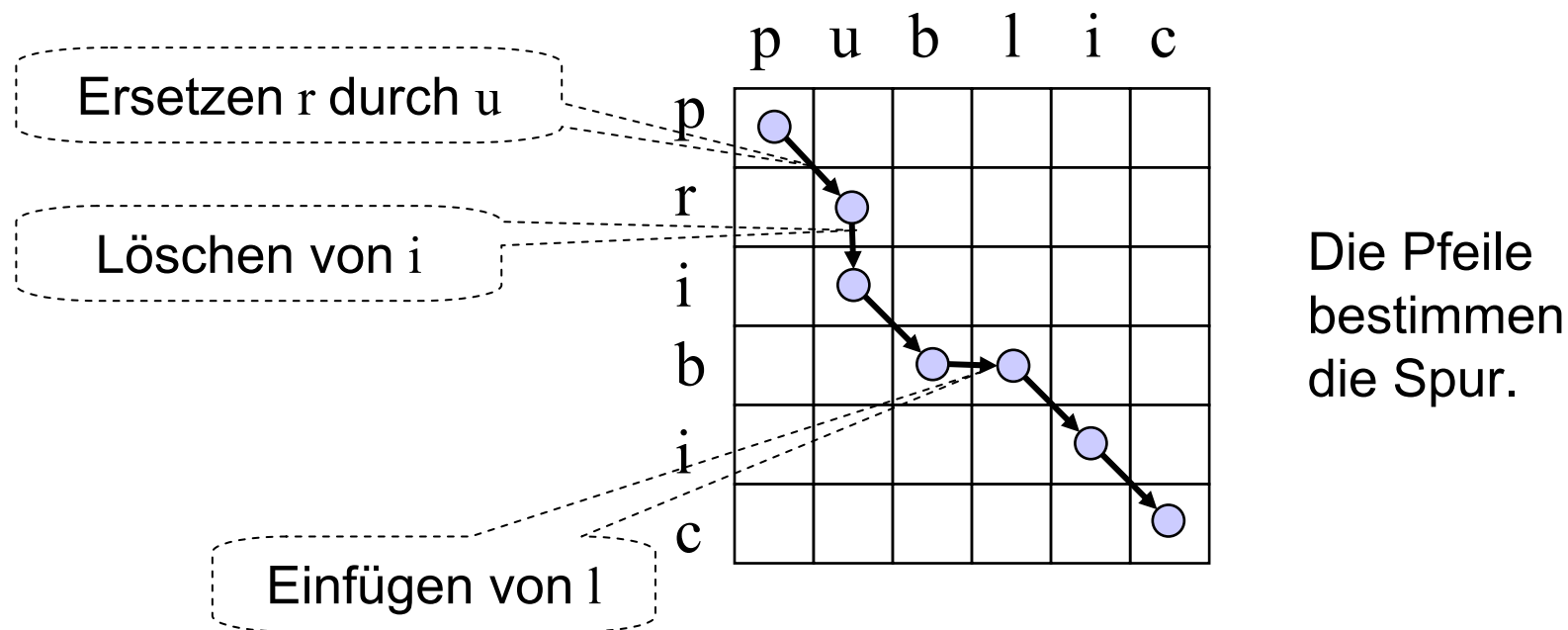


Eine Spur enthält keine kreuzenden Kanten!



Darstellung als Matrix:

↘ Ersetzen
→ Hinzufügen
↓ Löschen



Bestimmung der Editierdistanz $C(A,B)$ mittels dynamischen Programmierens

- Löse wiederum Gesamtproblem durch Lösen und Zwischenspeichern von Teilproblemen
- Gesucht ist $C(a_1 \dots a_n, b_1 \dots b_m)$ von zwei Zeichenketten $a_1 \dots a_n$ und $b_1 \dots b_m$

Gegeben: $A = a_1, a_2, \dots, a_n$ und $B = b_1, b_2, \dots, b_m$

Annahme:

Wir wissen, was die minimale Editierdistanz

- zwischen a_1, a_2, \dots, a_{i-1} und b_1, b_2, \dots, b_{j-1} sowie
- zwischen a_1, a_2, \dots, a_{i-1} und b_1, b_2, \dots, b_j sowie
- zwischen a_1, a_2, \dots, a_i und b_1, b_2, \dots, b_{j-1} ist.

		b_{j-1}	b_j	
a_{i-1}				
a_i			?	

Wie berechnet sich die minimale Editierdistanz $C(i,j)$

zwischen a_1, a_2, \dots, a_i und b_1, b_2, \dots, b_j ?

Berechne $C_{i,j}$ aus
 $C_{i-1,j}$, $C_{i,j-1}$ und $C_{i-1,j-1}$!

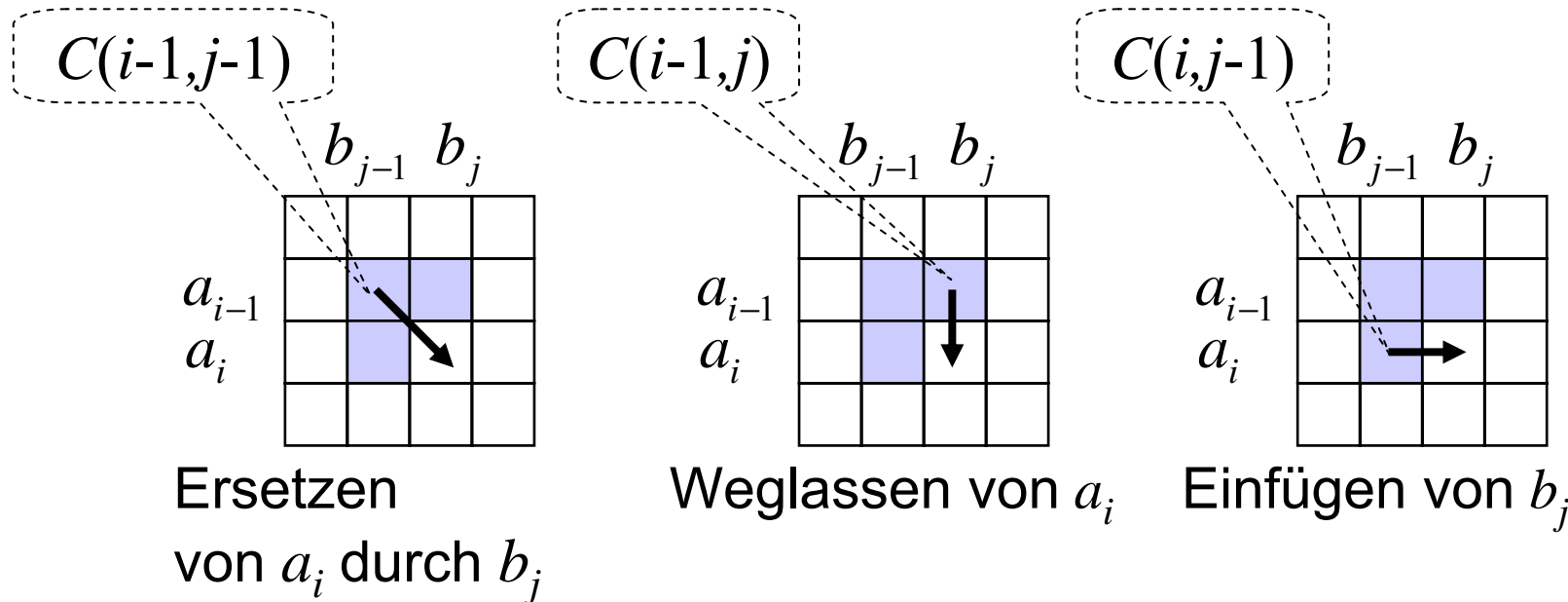


Bestimmung der Editierdistanz $C(A,B)$ mittels dynamischen Programmierens

- Versuche aus Menge von Teillösungen $C_{i,j}(a_1 \dots a_i, b_1 \dots b_j)$, $0 \leq i \leq n$, $0 \leq j \leq m$ Gesamtlösung $C(a_1 \dots a_n, b_1 \dots b_m)$ zu konstruieren.
- Zunächst gilt als Voraussetzung:
 - $C_{0,0} = C(\varepsilon, \varepsilon) = 0$ Leere Wörter
 - $C_{0,j} = C(\varepsilon, b_1 \dots b_j) = j$, $1 \leq j \leq m$ j Einfüge-Operationen
 - $C_{i,0} = C(a_1 \dots a_i, \varepsilon) = i$, $1 \leq i \leq n$ i Lösch-Operationen
- Wie bestimmt sich ein $C_{i,j}$ aus $C_{i-1,j}$, $C_{i,j-1}$ und $C_{i-1,j-1}$?



Aktualisieren der mitgeführten (akkumulierten) Editierdistanz:



$$C_s(i, j) = C(i-1, j-1) + \begin{cases} 0 & \text{falls } a_i = b_j \\ 1 & \text{falls } a_i \neq b_j \end{cases}$$

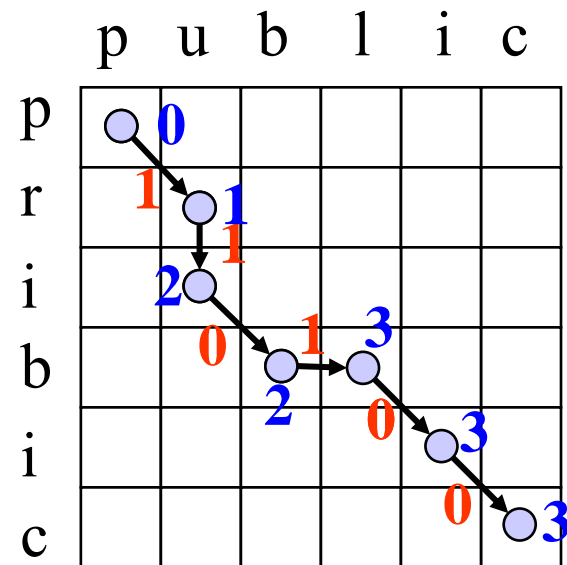
$$C_d(i, j) = C(i-1, j) + 1$$

$$C_i(i, j) = C(i, j-1) + 1$$

$$\text{also } C(i, j) = \min \begin{cases} C_s(i, j) \\ C_d(i, j) \\ C_i(i, j) \end{cases}$$



Im Beispiel:



- Editierdistanz ist in der rechten unteren Ecke der Matrix ablesbar



Vorausberechnung der Distanzschrirte und Speicherung in einer Matrix

$$C_s(i,j) = C(i-1,j-1) + \begin{cases} 0 & \text{falls } a_i = b_j \\ 1 & \text{falls } a_i \neq b_j \end{cases}$$

$$C_d(i,j) = C(i-1,j) + 1$$

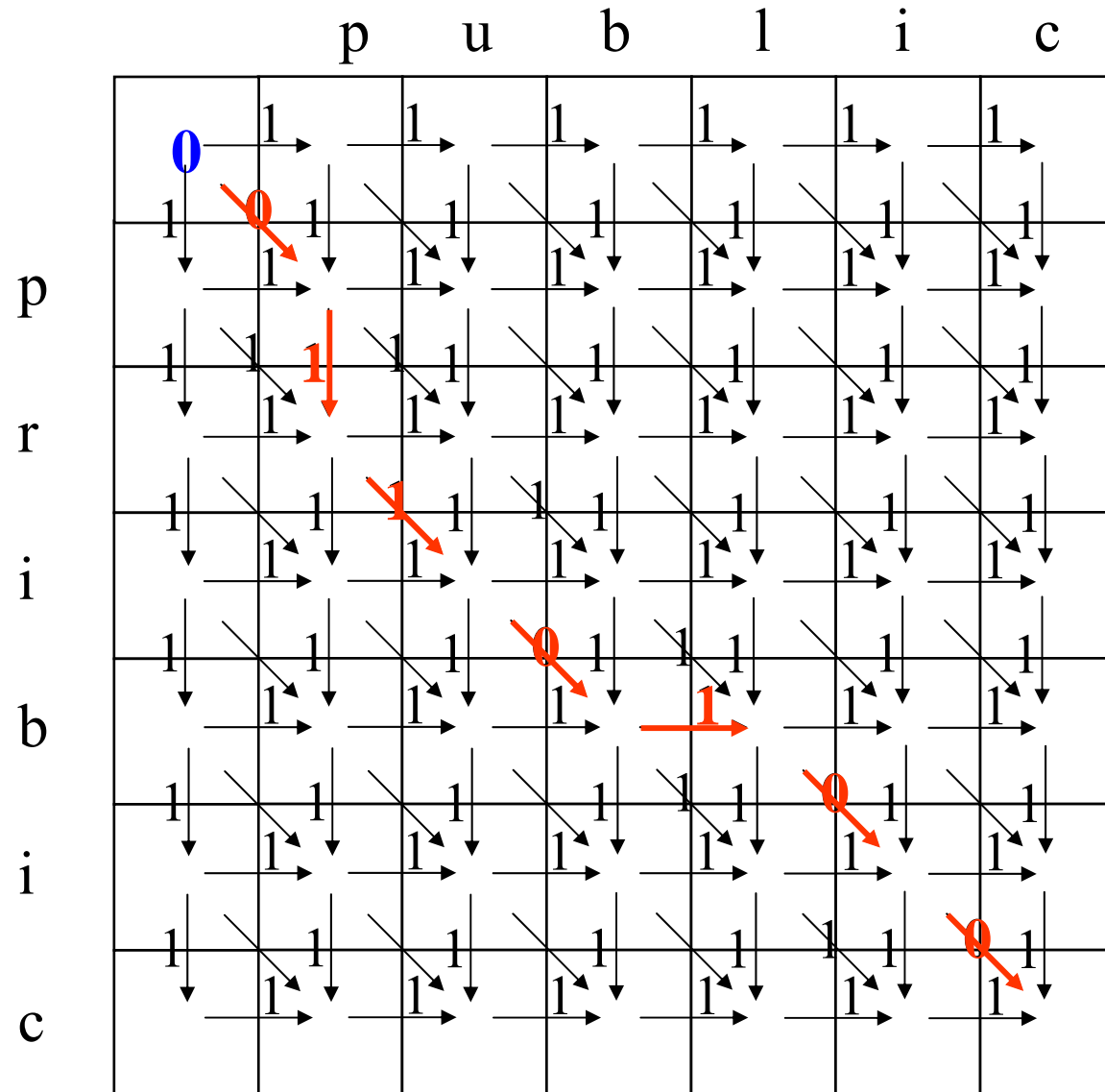
$$C_i(i,j) = C(i,j-1) + 1$$

Spezialfall Anfang

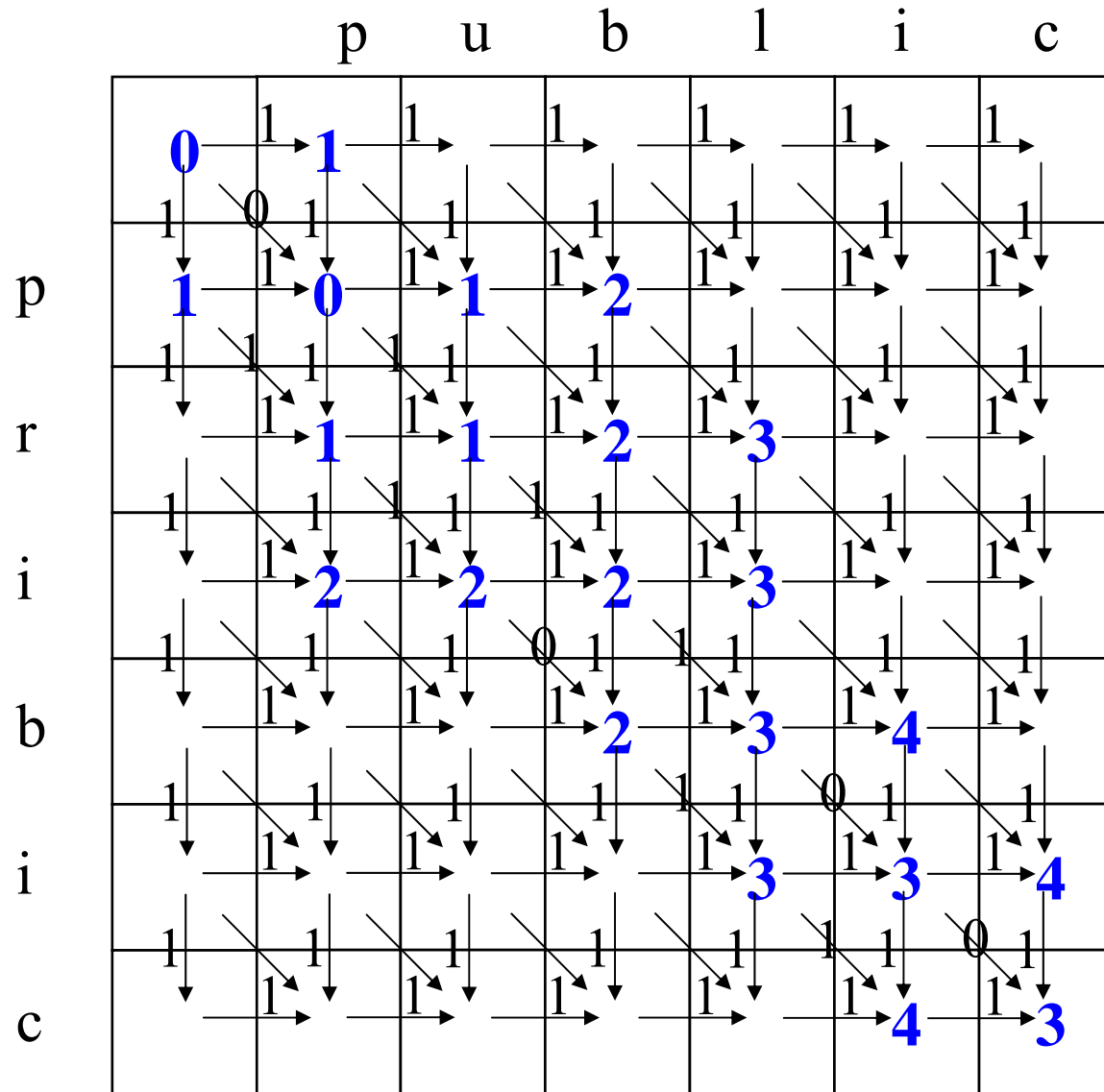
- Wie erfasst man, wenn im ersten Zeichen keine Übereinstimmung herrscht, also bereits dort Ersetzen, Weglassen oder Einfügen erforderlich ist?
- Definiere $C(i,0)=C(0,j)=1 \ \forall i,j \neq 0$ sowie $C(0,0)=0$.



Beispiel $C(0,0)=0$



Minimale
Distanzen:



$$C(i,j) = \min \begin{cases} C_s(i,j) \\ C_d(i,j) \\ C_i(i,j) \end{cases}$$



Wie findet man jetzt die Spur?

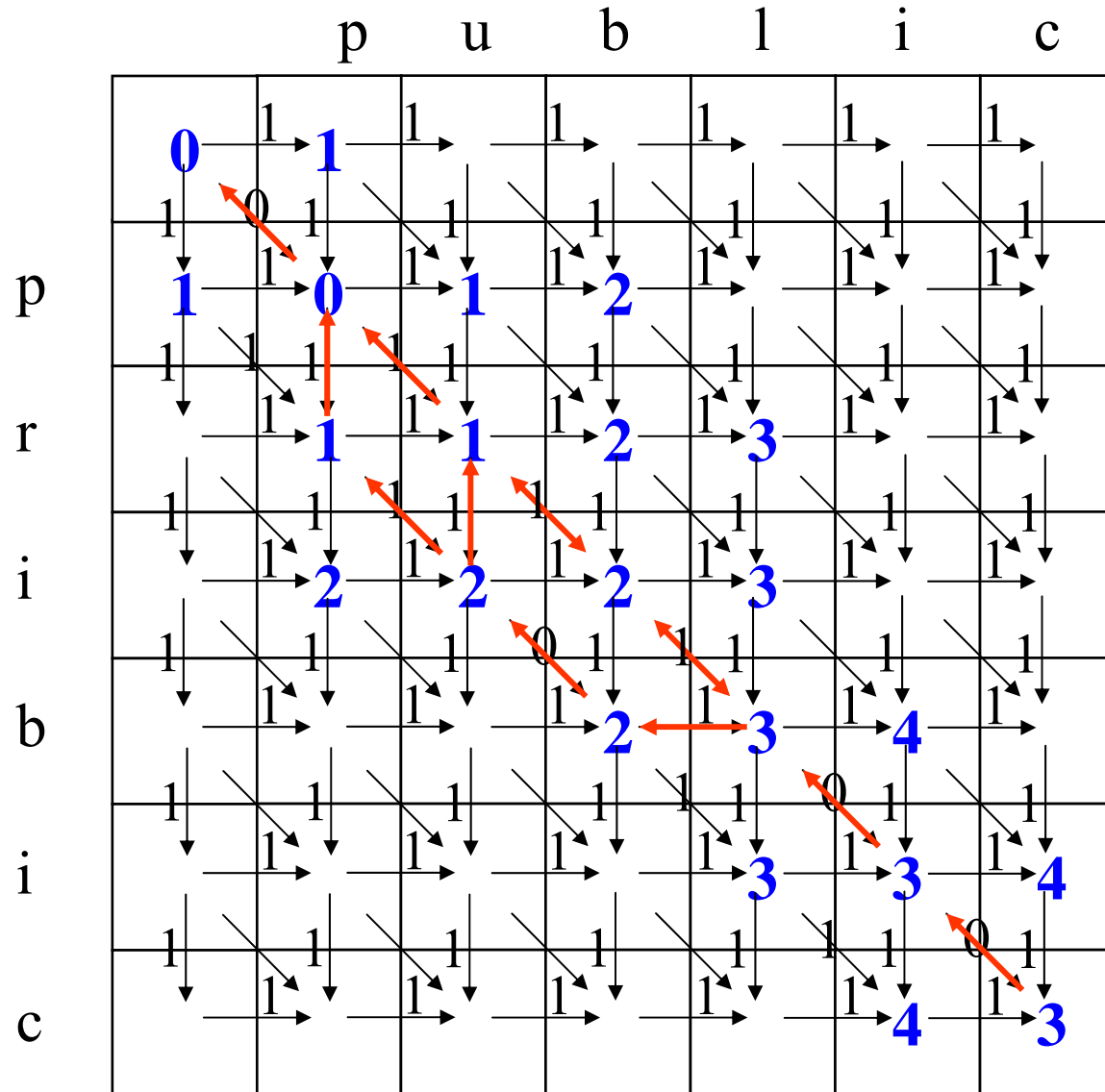
Spur:

- $(n,m), M(n,m), M(M(n,m)), \dots, M(M(\dots(n,m) \dots))$ mit $M(i,j)$ beste Vorgängerzelle von (i,j)
- $C_{\text{vorgänger}}$ minimal

Also: Zurücklaufen von (n,m) Richtung $(0,0)$ entlang dieser Bedingungen!



Minimale
Distanzen:



Mehrere
Möglichkeiten!



Gegeben: Text $A=a_1 \dots a_n$, Muster $B=b_1 \dots b_m$, $k \geq 0$

Gesucht: Alle Vorkommen von Muster B' in A mit $C(B, B') \leq k$

- Erste Idee: naives Vorgehen
 - Betrachte $a_i a_{i+1} \dots a_{i'}$ und berechne $C(a_i a_{i+1} \dots a_{i'}, B)$
mit $1 \leq i \leq i' \leq n$
 - Falls $C(a_i a_{i+1} \dots a_{i'}, B) \leq k$, dann ist (approximatives) Vorkommen gefunden
- Nachteil Aufwand: $O(n^2 k m)$

Verändere Problemstellung leicht:

- Bestimme für jedes i im Text A das ähnlichste Teilstück, das bei i endet und die kleinste Editierdistanz zu B hat.
- Also: suche für jedes i mit $1 \leq i \leq n$, ein i' mit $1 \leq i' \leq i$, so dass für jedes i'' mit $1 \leq i'' \leq i$ gilt: $C(a_{i'} \dots a_i, B) \leq C(a_{i''} \dots a_i, B)$.



Damit können wir das Originalproblem lösen

- Berechne zu jeder Textstelle i nicht nur ein dort endendes B ähnliches Teilstück, sondern auch gleich die Editierdistanz dieses Teilstücks zu B .
- ⇒ Für jede Textstelle i kann überprüft werden, ob es überhaupt ein an i endendes Teilstück gibt, das eine Editierdistanz $\leq k$ hat.
- ⇒ Dann kennen wir ein an i endendes zu B ähnlichstes Teilstück von A

Verwende dafür die Lösung zum Berechnen der minimalen Editierdistanz.

- Erstelle $(m+1)(n+1)$ Matrix

$$C_{0,j}=0 \quad \text{für } 0 \leq j \leq n$$

$$C_{i,0}=i \quad \text{für } 0 \leq i \leq m$$

$C_s(i,j) = C(i-1,j-1) + \begin{cases} 0 & \text{falls } a_i = b_j \\ 1 & \text{falls } a_i \neq b_j \end{cases}$	$C_d(i,j) = C(i-1,j) + 1$
$C_i(i,j) = C(i,j-1) + 1$	

$$C(i,j) = \min \begin{cases} C_s(i,j) \\ C_d(i,j) \\ C_i(i,j) \end{cases}$$



- Gibt es in der Matrix einen Weg von $C_{0,j'-1}$ nach $C_{i,j}$
⇒ $a_{j'} \dots a_j$ ist ein zu $b_1 \dots b_i$ ähnlichstes und bei j endendes Teilstück in A
- Die Distanz dieses Teilstückes zu $a_{j'} \dots a_j$ ist $C_{i,j}$

Notwendige Editieroperationen

- Analog zur Berechnung Editierabstand mit Ändern, Löschen, Ersetzen
- Zum Nachvollziehen, welche Operationen notwendig sind, beginne wieder bei „unten-rechts“ rückwärts bis „oben-links“

Betrachte letzte Zeile!

- An jeder Stelle j gilt: hier passt das komplette Muster mit maximal $C_{m,j}$ Editierschritten in A
- Vergleiche $C_{m,j}$ mit k
 - wenn $C_{m,j} \leq k$, dann gesuchtes Muster B' gefunden, für das gilt $C(B, B') \leq k$
 - Und: B' endet an Stelle j in A



- Text A = adbb
- Muster B = abb, k = 1
- Initialisiere erste Zeile, erste Spalte
- Berechne die C(i,j)

↙ Ersetzen
→ Hinzufügen
↓ Löschen

		a	d	b	b
	0	0	0	0	0
a	1	0	1	1	1
b	2	1	1	1	1
b	3	2	2	1	1

$$C(i,j) = \min \begin{cases} C_s(i,j) \\ C_d(i,j) \\ C_i(i,j) \end{cases}$$

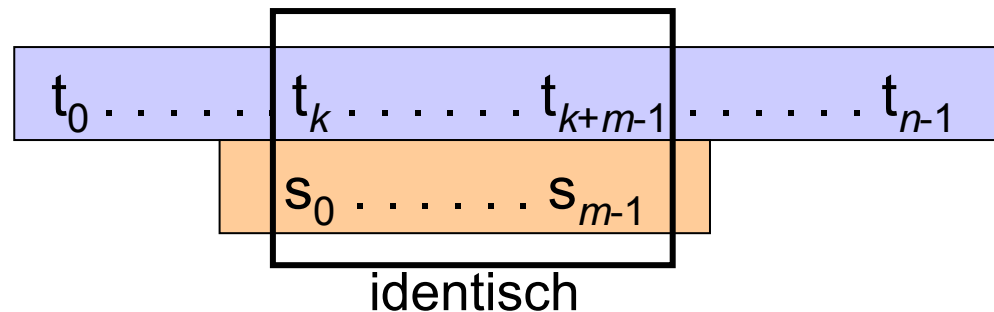
Hier passt das komplette Muster B mit $\leq k$.



13.3 Suchmuster in Texten

Gegeben: Text t vom Typ $\text{char}[n]$,
Suchmuster s vom Typ $\text{char}[m]$.

Gesucht: k , $0 \leq k \leq n - m$, so dass



falls k existiert, $k = -1$ andernfalls.

→ Hier interessiert uns jetzt die genaue Übereinstimmung!



Einfachste und anschauliche Lösung

- Schrittweises Verschieben des Musters und Vergleich:

```
static int textsuche(char[] t, char[] s) {  
    int n = t.length;  
    int m = s.length;  
    int j;  
  
    for (int k = 0; k <= (n-m); k++) {  
        for (j = 0; (j < m) && (t[k+j] == s[j]); j++);  
        if (j == m) return k;  
    }  
  
    return -1;  
}
```



Illustration ($n = 12$, $m=3$)

t =	F	a	h	r	r	a	d	l	a	d	e	n	k	j
s =	a	d	e										0	0
		a	d	e									1	1
			a	d	e								2	0
				a	d	e							3	0
					a	d	e						4	0
						a	d	e					5	2
							a	d	e				6	0
								a	d	e			7	0
									a	d	e	—	8	3



Aufwandsabschätzung: Ungünstigster Fall

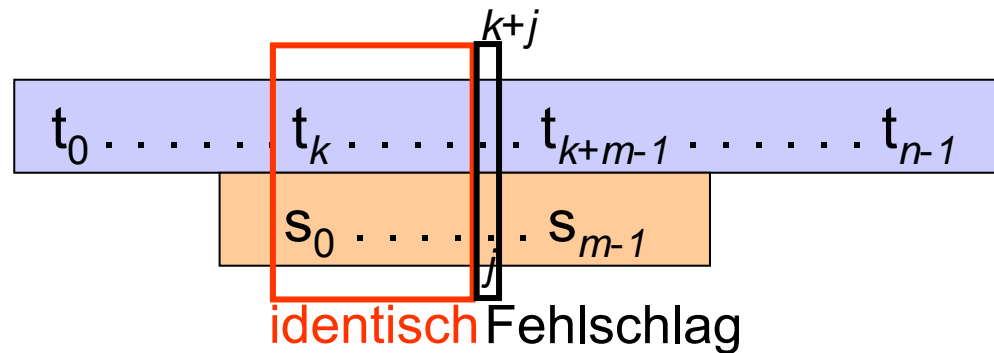
- Suche im Text bis zum Ende
- Mit jedem Schritt Vergleich bis zum Ende des Musters.
- Also Zahl der Vergleiche $O((n - m) * m) = O(m * n)$.

Verbesserung: Besitzt man Vorwissen über Text oder Muster, so kann man dieses in den Algorithmus einbringen, bevor die eigentliche Berechnung beginnt.



Überlegung zum Vorwissen

- Angenommen: Fehlschlag bei Position $k+j$:

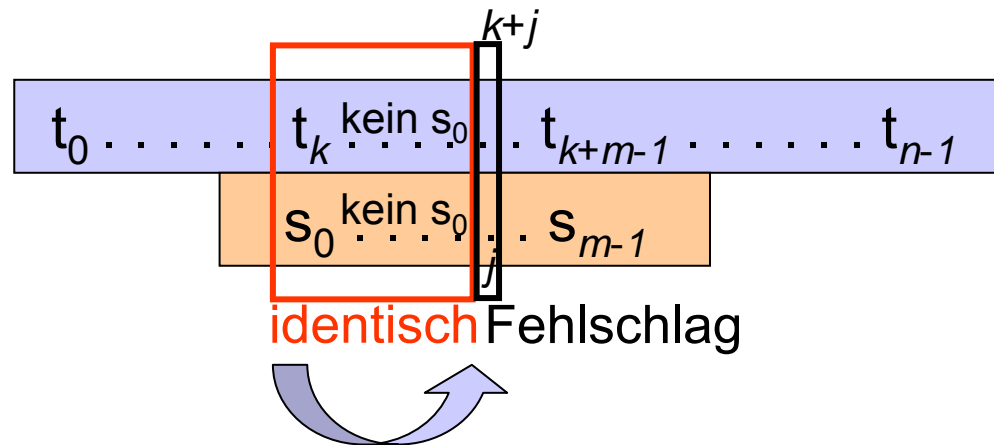


- Um wie viel kann man s verschieben, ohne einen Treffer (übereinstimmendes Zeichen) zu verpassen?



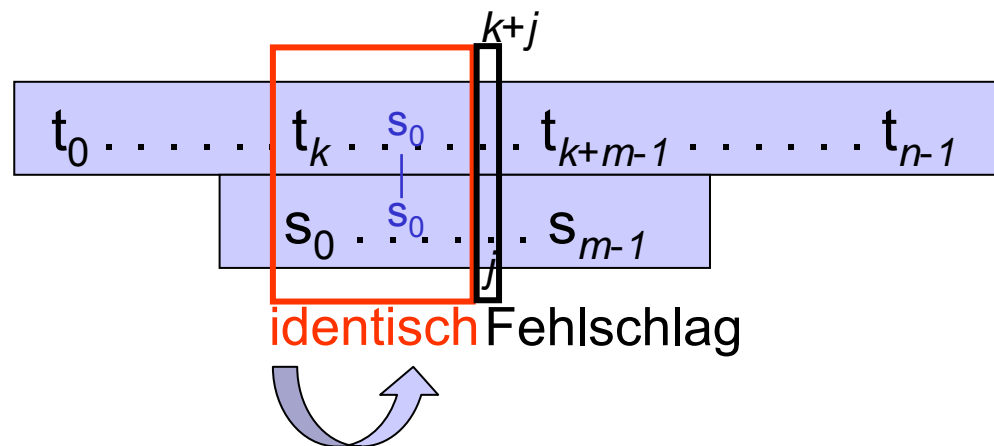
Fall 1: Für alle i , $0 < i < j$, $s[i] \neq s[0]$.

- Dann kommt $s[0]$ auch nicht in $t[p]$, $k < p < k+j$ vor, k kann an die Stelle $k+j$ vorrücken.



Fall 2: Es gibt i , $0 < i < j$, $s[i] = s[0]$.

- Dann braucht s erst wieder an der Stelle $k+i$ aufzusetzen, da dazwischen garantiert kein Treffer möglich ist. Es kann mehrere derartige i geben.
- Da aber von i bis $j-1$ Übereinstimmung herrschte, kommt nur ein i in Frage, für das in s als Wiederholungsfolge existiert: $s[i:j-1] = s[0:j-i-1]$.
- Also: Suche nach $i = \text{imin}(j)$ derart dass $0 < i < j$ und i minimal für $s[i:j-1] = s[0:j-i-1]$.



Allgemein:

- Setze auf an Stelle $k + \text{imin}(j)$. Für $j=0$ und $j=1$ setzen wir $\text{imin}(j)=1$, um das unbedingte Fortschalten um eine Position zu erfassen. Falls keine Wiederholung, setzen wir gemäß oben $\text{imin}(j) = j$.



a	b	r	a	k	a	d	a	b	r	a	a	b	r	a	c	a	d	a	b	r	a	k	j	imin
a	b	r	a	c	a	d	a	b	r	a												0	4	3
			a	b	r	a	c	a	d	a	b	r	a									3	1	1
				a	b	r	a	c	a	d	a	b	r	a								4	0	1
					a	b	r	a	c	a	d	a	b	r	a							5	1	1
						a	b	r	a	c	a	d	a	b	r	a						6	0	1
							a	b	r	a	c	a	d	a	b	r	a					7	4	3
										a	b	r	a	c	a	d	a	b	r	a		10	1	1
											a	b	r	a	c	a	d	a	b	r	a	11		



Vergleich mit vorhergehendem Beispiel (keine Wiederholung im Muster)

F	a	h	r	r	a	d	l	a	d	e	n	k	j	imin
a	d	e										0	0	1
	a	d	e									1	1	1
		a	d	e								2	0	1
			a	d	e							3	0	1
				a	d	e						4	0	1
					a	d	e					5	2	2
							a	d	e			7	0	1
								a	d	e	—	8		



Weitere Verbesserung:

- Wegen $i = \text{imin}(j)$ für $s[i:j-1] = s[0:j-i-1]$ gilt
 $s[0:j-\text{imin}-1] = s[\text{imin}:j-1] = t[k+\text{imin}:k+j-1]$
- Der Vergleich muss daher nicht mit Textposition $k+\text{imin}$ beginnen, sondern kann bei $j>0$ mit Textposition $k+j$ fortsetzen.
- Anders formuliert: Fortsetzung mit Musterposition
 $\text{shift}(j) := j - \text{imin}(j)$, $j>0$, $\text{shift}(0) = 0$.
- Das aber bedeutet:
 k kann mit jedem erfolgreichen Vergleich oder bei sofortigem Fehlschlag (Fehlschlag für $\text{shift}=0$) hochgezählt werden und bleibt sonst unverändert (k nicht-fallend).



a	b	r	a	k	a	d	a	b	r	a	a	b	r	a	c	a	d	a	b	r	a	k	j	imin
a	b	r	a	c	a	d	a	b	r	a												0	4	3
			a	b	r	a	c	a	d	a	b	r	a									3	4	1
				a	b	r	a	c	a	d	a	b	r	a								4	0	1
					a	b	r	a	c	a	d	a	b	r	a							5	1	1
						a	b	r	a	c	a	d	a	b	r	a						6	0	1
							a	b	r	a	c	a	d	a	b	r	a					7	4	3
										a	b	r	a	c	a	d	a	b	r	a		10	11	1
											a	b	r	a	c	a	d	a	b	r	a	11		



Da imin und daher shift nur von s und j abhängen, kann shift durch Vorausberechnung ermittelt werden.

j	0	1	2	3	4	5	6	7	8	9	10
$s[j]$	a	b	r	a	c	a	d	a	b	r	a
$\text{imin}(j)$	1	1	2	3	3	5	5	7	7	7	10
$\text{shift}(j)$	--	0	0	0	1	0	1	0	1	2	3



```
static int kmp(char[] t, char[] s) {  
    int[] shift = kmp_init(s);  
    int    k, j;  
  
    for ( k = 0, j = 0; k<t.length; ) {  
        if (j == s.length) {  
            return k-j;           // voller Treffer  
        } else if (t[k] == s[j] ) {  
            k++; j++;             // Zeichentreffer  
        } else if (j == 0) {  
            k++;                  // Fehlschlag am Anfang  
        } else j = shift[j];     // Fehlschlag mit  
                                   // neuer Musterposition  
    }  
    if (j == s.length) return k-j; // Anfangspos. im Text  
  
    return -1;  
}
```



Vorausberechnung von shift (`kmp_init`):

```
static int[] kmp_init(char[] s) {  
    int[] shift = new int [s.length+1];  
  
    shift[1] = 0;  
    for ( int k = 1, j = 0; k<s.length; ) {  
        if (s[k] == s[j] ) {  
            k++; j++; shift[k] = j;  
        } else if (j == 0) {  
            k++; shift[k] = j;  
        } else j = shift[j];  
    }  
  
    return shift;  
}
```

