Algorithmen und Datenstrukturen

Vorlesung 11

Dynamische Programmierung

UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Dynamische Programmierung (DP)

UNI FREIBURG

- Wichtige Algorithmenentwurf-Technik!
- Einfache, aber oft sehr effektive Idee

- Viele Probleme, welche naiv exponentielle Zeit benötigen, können mit dynamischer Programmierung in polynomieller Zeit gelöst werden.
  - Das gilt insbesondere für Optimierungsprobleme (min / max)

DP ≈ vorsichtige / optimierte Brute-Force-Lösung

DP ≈ Rekursion + Wiederverwendung

DP: Geschichte

- Woher kommt der Name?
- DP wurde durch Richard E. Bellman in den 1940er/1950er Jahren entwickelt. In seiner Autobiographie steht:

  "I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. … The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word research. … His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. … Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. … It also has a very interesting property as an adjective, and that it's impossible to use the word dynamic in a pejorative sense. … Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. …"

Fibonacci Zahlen

Definition der Fibonacci Zahlen $F_0, F_1, F_2, \ldots$:

$$F_0 = 0, F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

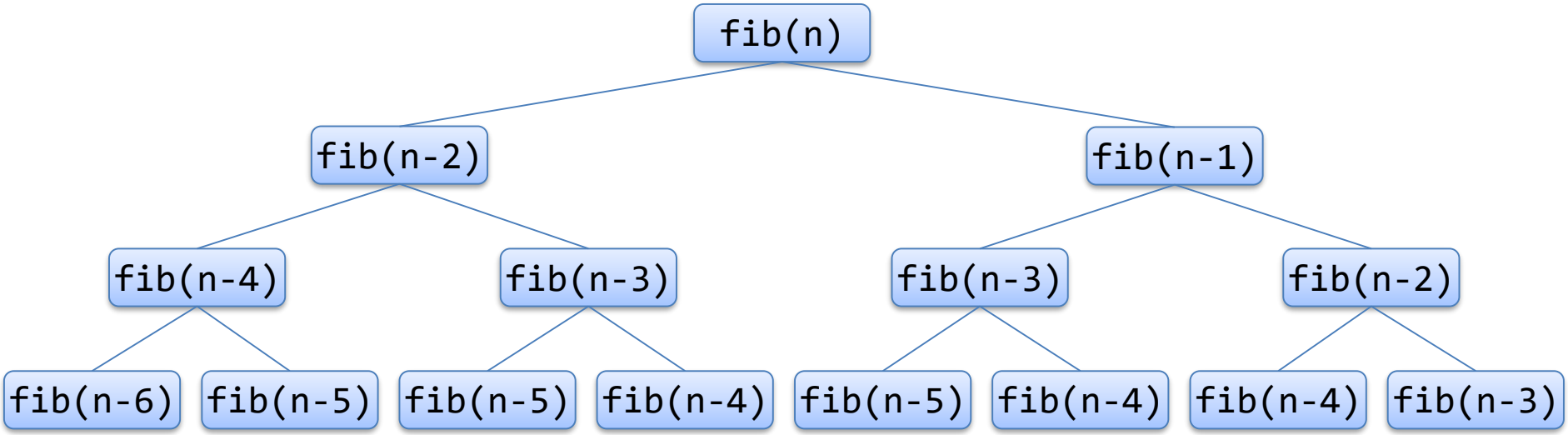
Ziel: Berechne F_n

- Das kann man einfach rekursiv tun

```
def fib(n):
    if n < 2:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    return f
```

Laufzeit Rekursiver Algorithmus

```
def fib(n):
    if n < 2:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    return f
```



- Rekursionsbaum ist Binärbaum, der bis zur Tiefe $n/2$ voll ist.
- Laufzeit : $\Omega\left(2^{n/2}\right)$
  - Wir berechnen aber immer wieder die gleichen Dinge!

Algorithmus mit Memoization

Memoization: Man merkt sich schon berechnete Werte (auf einem Notizzettel = memo)

```
memo = {}
def fib(n):
    if n in memo: return memo[n]
    if n < 2:
        f = n
    else:
        f = fib(n-1) + fib(n-2)
    memo[n] = f
    return f
```

erzeugt einen Dictionary (eine Hashtabelle)

Überprüfe zuerst, ob wir `fib(n)` schon berechnet haben.

Speichere berechneten Wert für `fib(n)` in der Hashtabelle.

- Jetzt wird jeder Wert `fib(i)` nur einmal rekursiv berechnet
  - Für jedes i gehen wir nur einmal durch den blauen Teil
  - Der Rekursionsbaum hat damit $\leq n$ innere Knoten.
  - Die Laufzeit ist deshalb $O(n)$.

DP: Etwas Genauer …

DP ≈ Rekursion + Memoization

Memoize: *Speichere* Lösungen zu *Teilproblemen*, verwende die gespeicherten Lösungen, falls das gleiche Teilproblem wieder auftaucht.

- Bei den Fibonacci-Zahlen sind die Teilprobleme $F_1, F_2, F_3, \ldots$

Laufzeit = #Teilprobleme · Zeit pro Teilprobleme

Normalerweise, einfach die Anzahl rekursive Aufrufe pro Teilproblem.

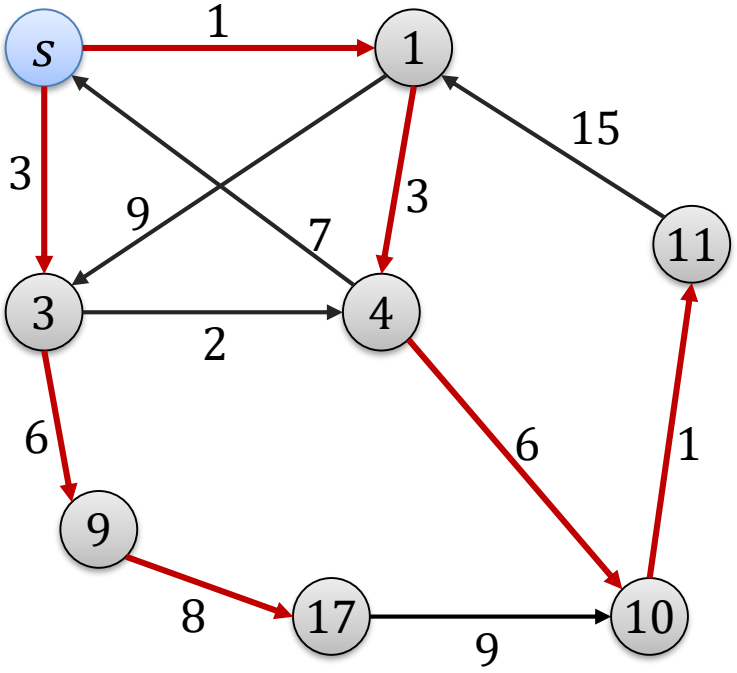
Fibonacci: Bottom-Up

```
def fib(n):
    fn = {}
    for k in [0,1, 2, …, n]:
        if k < 2:
            f = k
        else:
            f = fn[k-1] + fn[k-2]
        fn[k] = f
    return fn[n]
```

- Gehe durch die Teilprobleme in einer Reihenfolge, so dass man die Teilprobleme, die benötigt werden immer schon berechnet hat.
  - Im Fall der Fibonacci-Zahlen, berechne F_{i-2} und F_{i-1}, bevor F_i berechnet wird.

Kürzeste Wege mit DP

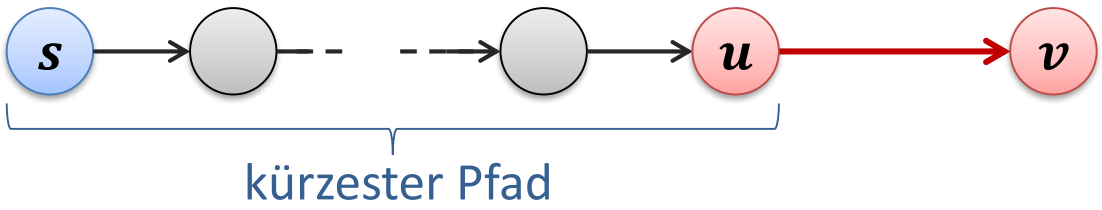
- **Gegeben:** gewichteter, gerichteter Graph $G = (V, E, w)$
  - Startknoten $s \in V$
  - Wir bezeichnen Gewicht einer Kante (u, v) als $w(u, v)$
  - Annahme: $\forall e \in E: w(e)$, keine negativen Kreise
- **Ziel:** Finde kürzeste Pfade / Distanzen von s zu allen Knoten
  - Distanz von s zu v: $d_G(s, v)$ (Länge eines kürzesten Pfades)



Kürzeste Wege, rekursive Formulierung

Rekursive Charakterisierung von $d_G(s,v)$?

- Wie sieht ein kürzester Pfad von s nach v aus?
- **Optimalität von Teilpfaden:**
  Falls $v \neq s$, dann gibt es einen Knoten u, so dass der kürzeste Pfad aus eine kürzesten Pfad von s nach u und der Kante (u,v) besteht.



$$\forall v \neq s \;:\; d_G(s,v) = \min_{u \in N_{\text{in}}(v)} d_G(s,u) + w(u,v)$$

- Können wir damit die Werte $d_G(s,v)$ rekursiv berechnen?

Kürzeste Wege, rekursive Formulierung

Rekursive Charakterisierung von $d_G(s,v)$?

$$d_G(s,v) = \min_{u \in N_{\text{in}}(v)} \{d_G(s,u) + w(u,v)\}, \qquad d_G(s,s) = 0$$

```
dist(v):
    d = ∞
    if v == s:
        d = 0
    else:
        for (u,v) in E:
            d = min(d, dist(u) + w(u,v))
    return d
```

Problem: Zyklen!

- Bei Zyklen erhalten wir eine unendliche Rekursion
  - Beispiel: Zyklus der Länge 2 (Kanten (u,v) und (v,u))
  - `dist(v)` ruft `dist(u)` auf, `dist(u)` ruft dann wieder `dist(v)` auf, etc.

Kürzeste Wege in azyklischen Graphen

```
memo = {}
dist(v):
    if v in memo: return memo[v]
    d = ∞
    if v == s:
        d = 0
    else:
        for (u,v) in E:    (gehe durch alle eingehenden Kanten von v)
        d = min(d, dist(u) + w(u,v))
    memo[v] = d
    return d
```

Laufzeit: $O(m)$

- Anzahl Teilprobleme: n
- Zeit für Teilproblem $d_G(s,v)$: #eingehende Kanten von v

Azyklische Graphen: Bottom-Up

Beobachtung:

- Kante $(u,v) \implies d_G(s,u)$ muss vor $d_G(s,v)$ berechnet warden
- Man kann die Knoten zuerst topologisch sortieren

Annahme:

- Reihenfolge $v_1, v_2, \dots, v_n$ ist topologische Sortierung der Knoten

```
D = “array of length n”
for i in [1:n]:
    D[i] = ∞
    if v_i == s:
        D[i] = 0
    else:
        for (v_j, v_i) in E:  (eingehende Kanten, top. Sortierung ⟹ j < i)
        D[i] = min(D[i], D[j] + w(v_j, v_i))
```

Kürzeste Wege in allgemeinen Graphen

UNI FREIBURG

Idee: Führe zusätzliche Teilprobleme ein, um die zyklischen Abhängigkeiten zu vermeiden

Teilprobleme $d_G^{(k)}(s,v)$

- Länge des kürzesten Pfades bestehend aus höchstens k Kanten

Rekursive Definition:

$$d_G^{(k)}(s,v) = \min\left\{d_G^{(k-1)}(s,v), \min_{(u,v)\in E}\left\{d_G^{(k-1)}(s,u) + w(u,v)\right\}\right\}$$

$$d_G^{(k)}(s,s) = 0, \qquad (\forall k \geq 0)$$

$$d_G^{(0)}(s,v) = \infty, \qquad (\forall v \neq s)$$

Kürzeste Wege in allgemeinen Graphen

UNI FREIBURG

```
memo = {}
dist(k, v):
    if (k, v) in memo: return memo[(k, v)]
    d = ∞
    if s == v:
        d = 0
    elif k > 0:
        d = dist(k-1, v)
        for (u,v) in E:
                                (gehe durch alle eingehenden Kanten von v)
            d = min(d, dist(k-1, u) + w(u,v))
    memo[(k, v)] = d
    return d

distance(v):
    return dist(n-1, v)
```

Kürzeste Wege mit DP: Laufzeit

Laufzeit bei DP typischerweise:

$$\#\text{Teilprobleme} \cdot \text{Zeit pro Teilproblem}$$

- Zeit pro Teilproblem: rekursive Aufrufe kosten 1 Zeiteinheit
  - Durch die Memoization wird jedes Teilproblem nur einmal aufgerufen
  - Rekursive Kosten sind daher durch 1. Faktor abgedeckt
- Zeit pro Teilproblem: typischerweise #rek. Möglichkeiten

Kürzeste Wege:

- #Teilprobleme: $O(n^2)$
- Zeit pro Teilproblem: #eingehende Kanten

Laufzeit: $O(m \cdot n)$

- Gleiche Laufzeit wie bei Bellman-Ford
  - Algorithmus entspricht im Wesentlichen dem Bellman-Ford Algorithmus.

Kürzeste Wege: Bottom-Up

UNI FREIBURG

- Normalerweise werden dynamische Programme bottom-up aufgeschrieben
  - ist oft effizienter (keine Rekursion, keine Hashtabelle)
  - ist oft eine natürliche Formulierung des Algorithmus
- Bottom-Up DP Algorithmus
  - Benötigt Reihenfolge in welcher die Teilprobleme berechnet werden können (topologische Sortierung des Abhängigkeitsgraphen)
  - Da man sowieso sicherstellen muss, dass keine zyklischen Abhängigkeiten bestehen, ist diese topologische Sortierung oft sehr einfach zu erhalten
- Reihenfolge beim Kürzeste Wege Problem
  - Sortiere $d_G^{(k)}(s,v)$ aufsteigend nach k
  - Für gleiche k-Werte gibt es keine Abhängigkeiten

Kürzeste Wege: Bottom-Up

```
dist = “2-dimensional array”
for k in range(n):
    for v in V:
        d = ∞
        if v == s:
            d = 0
        elif k > 0:
            d = dist[k-1, v]
            for (u,v) in E:
                                (gehe durch alle eingehenden Kanten von v)
                d = min(d, dist[k-1, u] + w(u,v))
        dist[k, v] = d
```

5 Schritte zur DP Lösung

UNI FREIBURG

5 Schritte	Analyse
1) Teilprobleme definieren	#Teilprobleme zählen
2) Raten (Teil der Lösung)	#Möglichkeiten zählen
3) Rekursionsformel aufstellen	Zeit pro Teilproblem
4) Rekursion + Memoization oder Bottom-Up DP Tabelle aufbauen	Zeit = Zeit pro Teilproblem · #Teilprobleme
5) Löse ursprüngliches Problem	Benötigt evtl. zusätzliche Zeit

- Dynamische Programmierung ist dann ein guter Ansatz, wenn man ein Problem rekursiv lösen kann, wenn aber die Anzahl der verschiedenen Teilprobleme, die man rekursiv lösen muss, nicht allzu gross ist.

5 Schritte zur DP Lösung

5 Schritte	Fibonacci-Zahl F_n
1) Teilprobleme definieren	#Teilprobleme $= n$
2) Raten (Teil der Lösung)	nichts zu raten, #Möglichkeiten $= 1$
3) Rekursionsformel aufstellen	Zeit pro Teilproblem $= O(1)$
4) Rekursion + Memoization oder Bottom-Up DP Tabelle aufbauen	Zeit $=$ Zeit pro Teilproblem $\cdot$ #Teilprobleme $= O(1) \cdot n = O(n)$
5) Löse ursprüngliches Problem	Lösung ist Teilproblem F_n, Zeit $O(1)$

5 Schritte	Single Source Shortest Paths (Bellman-Ford)
1) Teilprobleme definieren	#Teilprobleme $= n \cdot (n-1)$ (alle $d_G^{(k)}(s,v)$)
2) Raten (Teil der Lösung)	$d_G^{(k)}(s,v)$: Kante zu v, #Mögl.: $1 +$ Eingangsgrad von v
3) Rekursionsformel aufstellen	Zeit pro Teilproblem $= \Theta\big(1 + \text{in\_degree}(v)\big)$
4) Rekursion + Memoization oder Bottom-Up DP Tabelle aufbauen	Zeit $= \sum_{\text{Teilprobleme}}$ Zeit pro Teilproblem $= \sum_{v \in V} \Theta\big(1 + \text{in\_degree}(v)\big) = \Theta(\lvert V\rvert \cdot \lvert E\rvert)$
5) Löse ursprüngliches Problem	Alle $d_G^{(n-1)}(s,v)$, Zeit $O(\lvert V\rvert)$

Berechnen der Lösung

UNI FREIBURG

Rekursive Berechnung der Optimierungsfunktion

- Alle Möglichkeiten werden (rekursive) durchprobiert
- Die beste (min/max) wird ausgewählt

Berechnen der Lösung

- Der rekursive Aufruf der Optimierungsfunktion gibt nur den optimalen Funktionswert zurück (z.B. Länge des kürzesten Pfades)
- Um die rekursive berechnete Lösung zu erhalten, muss man sich merken, welche der Möglichkeiten in jedem Schritt den optimalen Wert ergeben hat
- Wenn man DP mit der Hashfunktion macht, speichert man diese Information auch in der Hashtabelle
- Bottom-Up: Man speichert in jede Zelle der Tabelle nicht nur den besten Wert, sondern auch wie man ihn erhalten hat

Lösung berechnen: Parent Pointers

UNI FREIBURG

Allgemeines DP

```
memo = {}
parent = {}
DP(x1, x2, …, xk):
    if (x1, x2, …, xk) in memo:
        return memo[(x1, x2, …, xk)]
    if (x1, x2, …, xk) in Basis
        value = …
    else:
        value = min/max des Wertes von DP(x1, x2, …, xk)
                über Vorgängerknoten (y1, y2, …, yk) im
                Abhängigkeitsgraphen
    memo[(x1, x2, …, xk)] = value
    parent[(x1, x2, …, xk)] = (y1, y2, …, yk)-Tupel, welches
                              das min/max erreicht hat
    return value
```

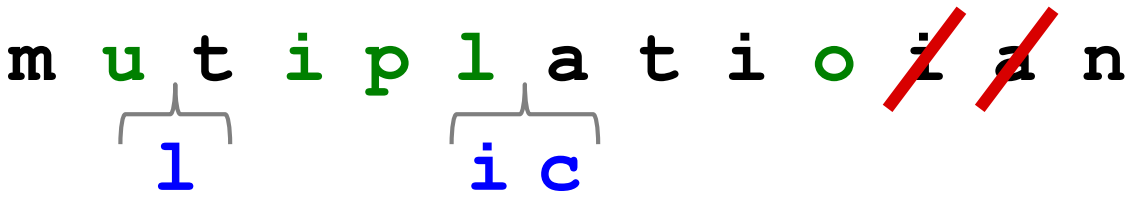
Editierdistanz

- Für zwei Zeichenketten A und B, berechne

  Editierdistanz $D(A,B)$ (# Editierop., um A in B zu überführen)

  und auch eine minimale Sequenz von Operationen, die A in B überführt.

- **Beispiel:** mathematician → multiplication:



Editierdistanz

Given: Two strings $A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$

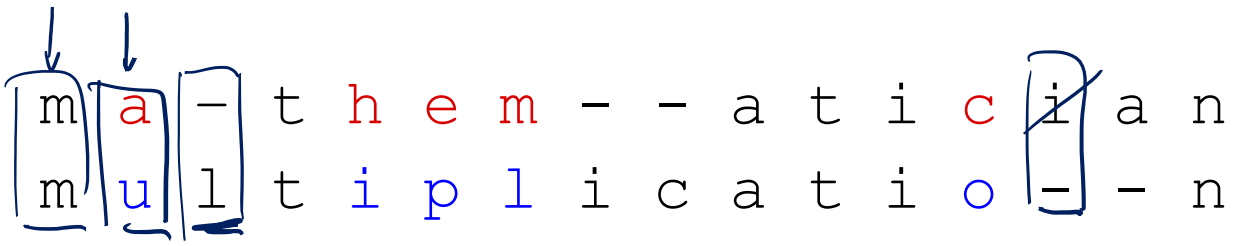
Goal: Determine the minimum number $D(A, B)$ of edit operations required to transform A into B

Edit operations:

a) **Replace** a character from string A by a character from B

b) **Delete** a character from string A

c) **Insert** a character from string B into A



Editierdistanz - Kostenmodell

- Cost for **replacing** character a by b: $\boldsymbol{c(a,b) \geq 0}$

- Capture insert, delete by allowing $a = \varepsilon$ or $b = \varepsilon$:
  - Cost for **deleting** character a: $\boldsymbol{c(a,\varepsilon)}$
  - Cost for **inserting** character b: $\boldsymbol{c(\varepsilon,b)}$

- **Triangle inequality**:

$$c(a,c) \leq c(a,b) + c(b,c)$$

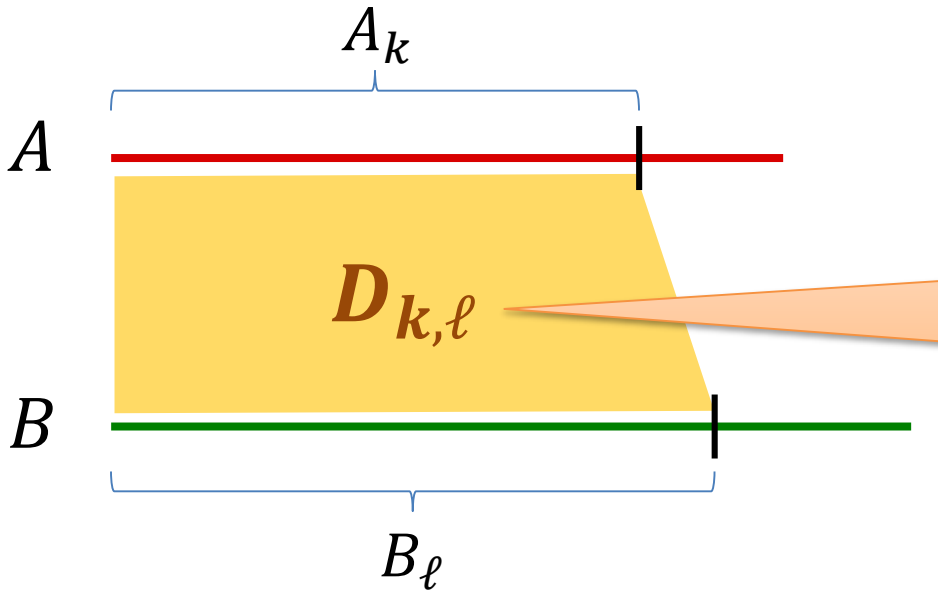
→ each character is changed at most once!

- **Unit cost model**: $c(a,b) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \end{cases}$

Editierdistanz: Teilprobleme

Definiere $A_k := a_1 \dots a_k$, $B_\ell := b_1 \dots b_\ell$

Teilprobleme: $D_{k,\ell} := D(A_k, B_\ell)$



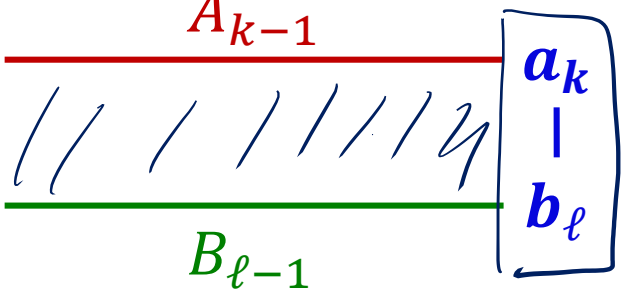
Editierdistanz zwischen Präfix A_k von A und Präfix B_ℓ von B

Berechnen der Editierdistanz

Three ways to end optimal "alignment" between A_k and B_ℓ:

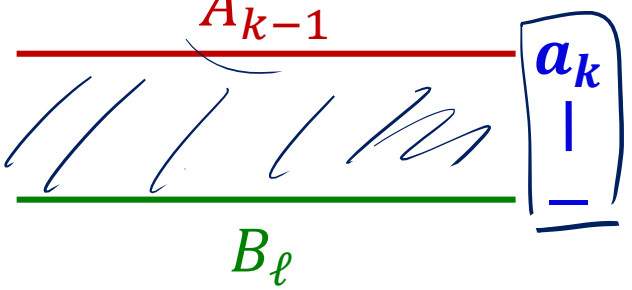
1. a_k is replaced by b_ℓ:

$$D_{k,\ell} = D_{k-1,\ell-1} + c(a_k, b_\ell)$$



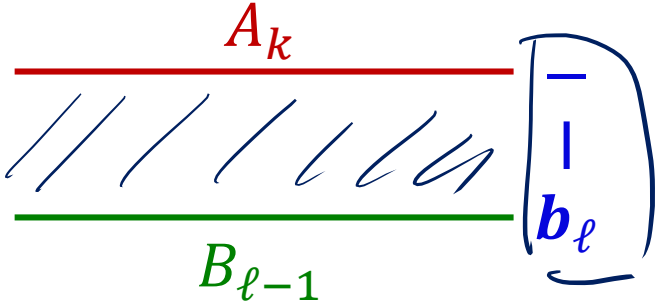
2. a_k is deleted:

$$D_{k,\ell} = D_{k-1,\ell} + c(a_k, \varepsilon)$$



3. b_ℓ is inserted:

$$D_{k,\ell} = D_{k,\ell-1} + c(\varepsilon, b_\ell)$$

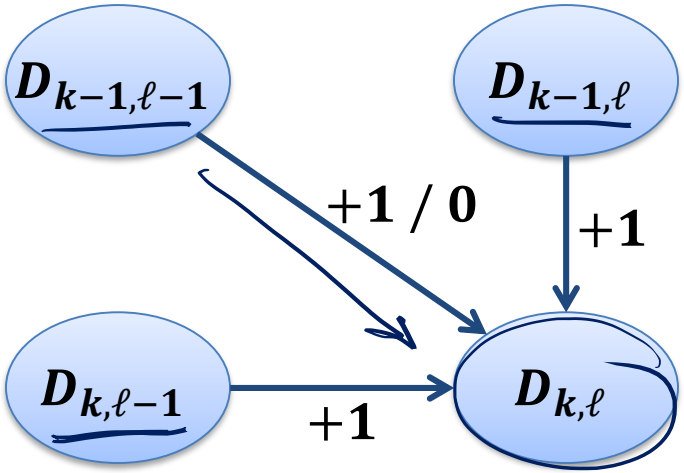


Berechnen der Editierdistanz

- Recurrence relation (for $k, \ell \geq 1$)

$$D_{k,\ell} = \min \begin{Bmatrix} D_{k-1,\ell-1} + c(a_k, b_\ell) \\ D_{k-1,\ell} + c(a_k, \varepsilon) \\ D_{k,\ell-1} + c(\varepsilon, b_\ell) \end{Bmatrix} = \underbrace{\min \begin{Bmatrix} D_{k-1,\ell-1} + 1 \,/\, 0 \\ D_{k-1,\ell} + 1 \\ D_{k,\ell-1} + 1 \end{Bmatrix}}_{\textbf{unit cost model}}$$

- Need to compute $D_{i,j}$ for all $0 \leq i \leq k$, $0 \leq j \leq \ell$:



Rekursionsgleichung Editierdistanz

Base cases:

$$\begin{aligned} D_{0,0} &= D(\varepsilon,\varepsilon) &&= 0 \\ D_{0,j} &= D(\varepsilon,B_j) = D_{0,j-1} + c(\varepsilon,b_j) &&= j \\ D_{i,0} &= D(A_i,\varepsilon) = D_{i-1,0} + c(a_i,\varepsilon) &&= i \end{aligned}$$

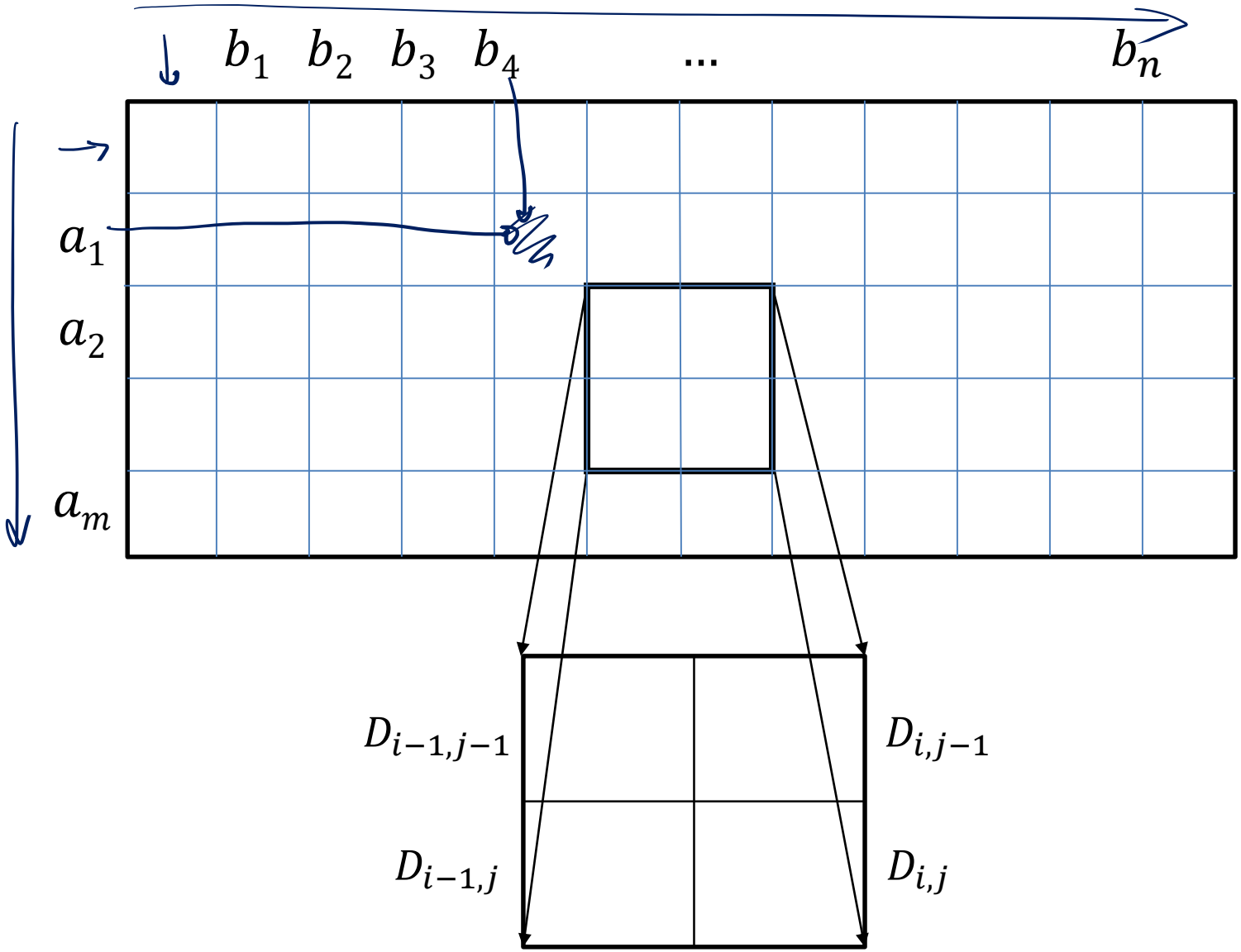
unit cost model

Recurrence relation:

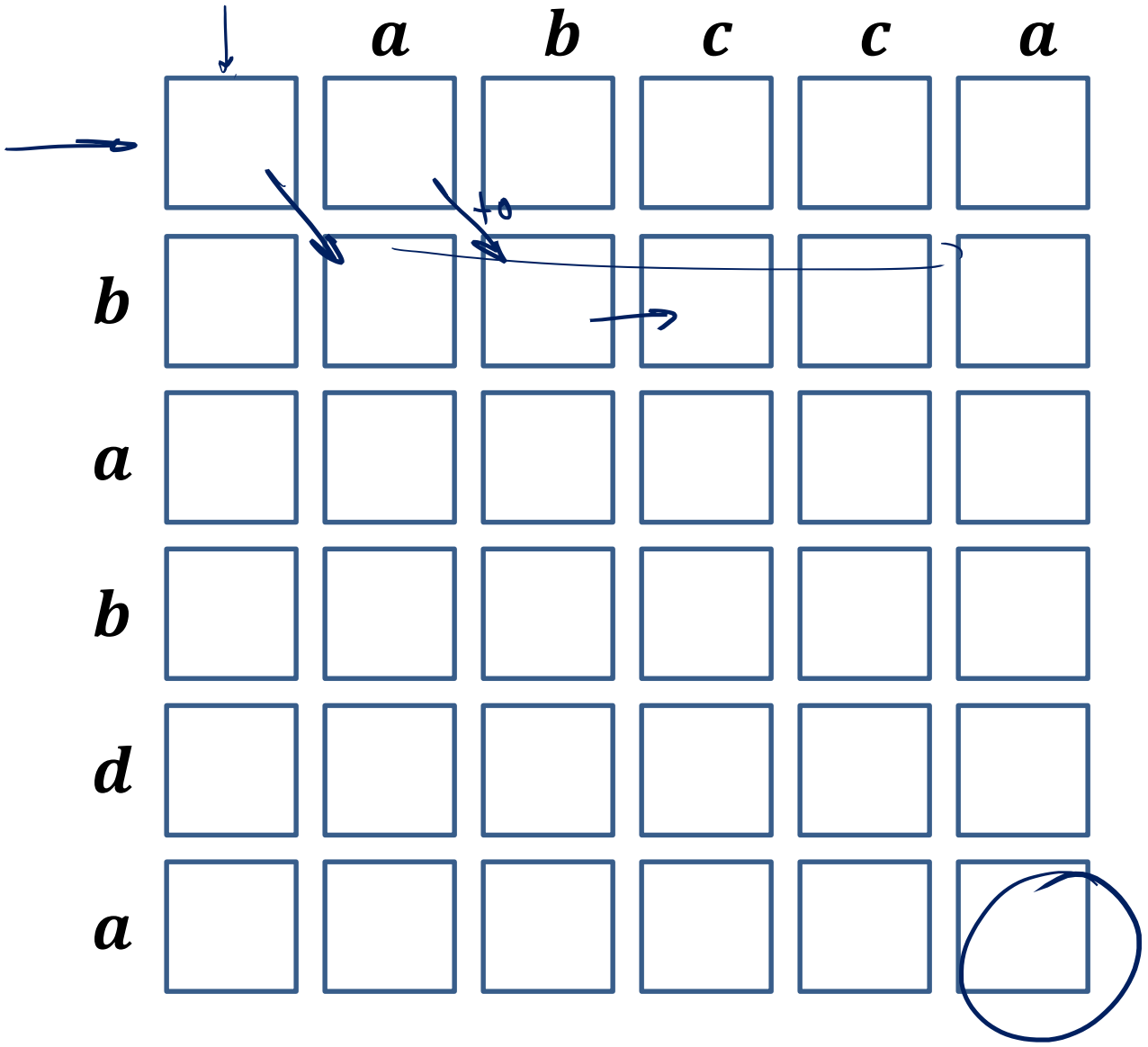
$$D_{i,j} = \min\begin{Bmatrix} D_{k-1,\ell-1} + c(a_k,b_\ell) \\ D_{k-1,\ell} + c(a_k,\varepsilon) \\ D_{k,\ell-1} + c(\varepsilon,b_\ell) \end{Bmatrix} = \min\begin{Bmatrix} D_{k-1,\ell-1} + 1\,/\,0 \\ D_{k-1,\ell} + 1 \\ D_{k,\ell-1} + 1 \end{Bmatrix}$$

unit cost model

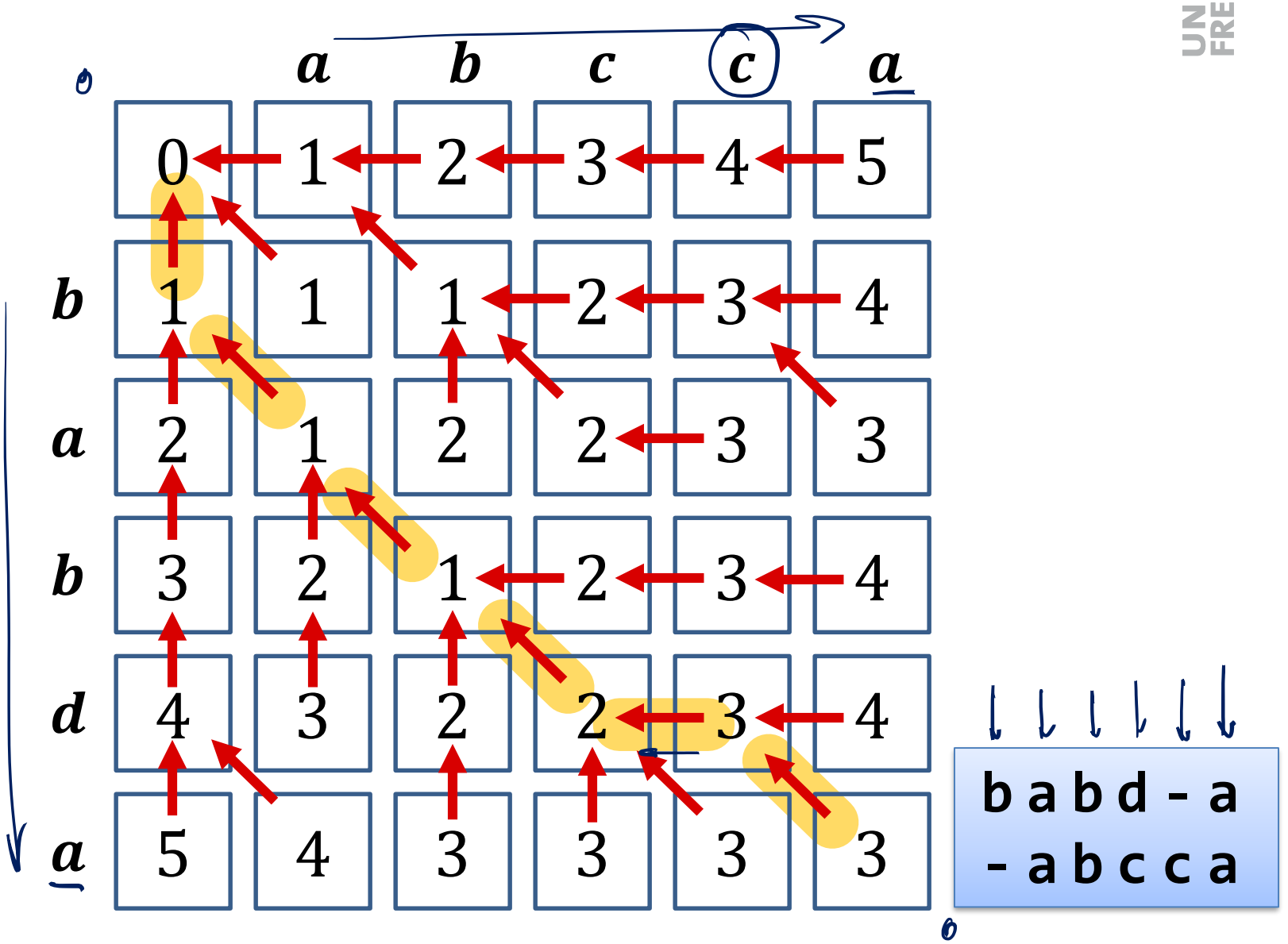
Reihenfolge der Teilprobleme



Beispiel

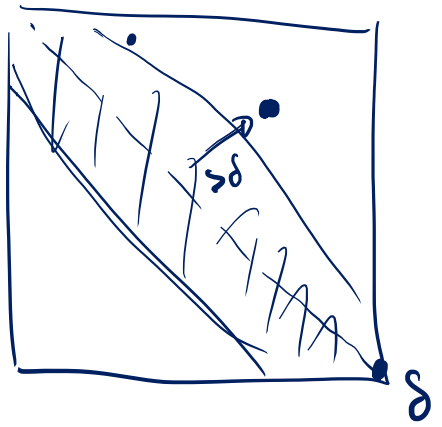


Editieroperationen



Editierdistanz – Zusammenfassung

- **Running Time:**
  - Edit distance between two strings of lengths m and n can be computed in $O(m \cdot n)$ time.
- **Obtain the edit operations:**
  - for each cell, store which rule(s) apply to fill the cell
  - track path backwards from cell (m, n)
- **Unit cost model:**
  - interesting special case, each edit operation costs 1



- **Optimization:**
  - If the edit distance is small, we do not need to fill out the whole table.
  - If the edit distance is $\leq \delta$, only entries at distance at most δ from the main diagonal of the table are really relevant.
  - For two strings of length n, we then only have to fill out $O(\delta \cdot n)$ entries.
  - With this idea, one can compute the edit distance in time $O\big(n \cdot D(A, B)\big)$.