ETH zürich

Departement Informatik

Programmieren und Problemlösen

Dynamische Programmierung

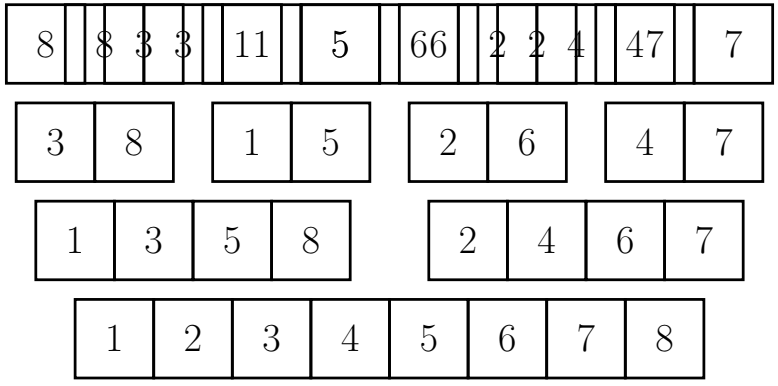
Dennis Komm

Frühling 2021 – 6. Mai 2021

Rekursives Sortieren und Suchen

$\mathcal{O}(n \log_2 n)$-Sortierer

Iteratives Mergesort

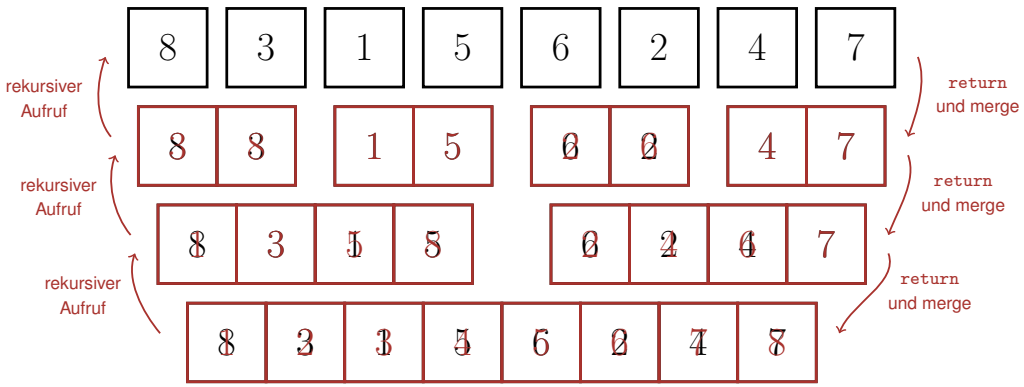


Iteratives Mergesort

Herzstück von Mergesort ist die Funktion `merge()`, welche zwei sortierte Listen zusammenfügt

```
def merge(leftdata, rightdata):
    result = []
    while len(leftdata) > 0 and len(rightdata) > 0:
        if leftdata[0] > rightdata[0]:
            result.append(rightdata.pop(0))
        else:
            result.append(leftdata.pop(0))
    return result + leftdata + rightdata
```

Rekursives Mergesort



Rekursives Mergesort

Mergesort als rekursive Python-Funktion

- die eine Liste als Parameter übergeben erhält
- diese in der Mitte in zwei Listen unterteilt
- den Algorithmus auf diese rekursiv anwendet
- die zwei so sortierten Listen „merged“ und zurückgibt

Rekursives Mergesort

```
def mergesort(data):
    if len(data) <= 1:
        return data
    mid = len(data) // 2
    leftdata = mergesort(data[:mid])
    rightdata = mergesort(data[mid:])
    result = []
    while len(leftdata) > 0 and len(rightdata) > 0:
        if leftdata[0] > rightdata[0]:
            result.append(rightdata.pop(0))
        else:
            result.append(leftdata.pop(0))

    return result + leftdata + rightdata
```


Rekursives Sortieren und Suchen

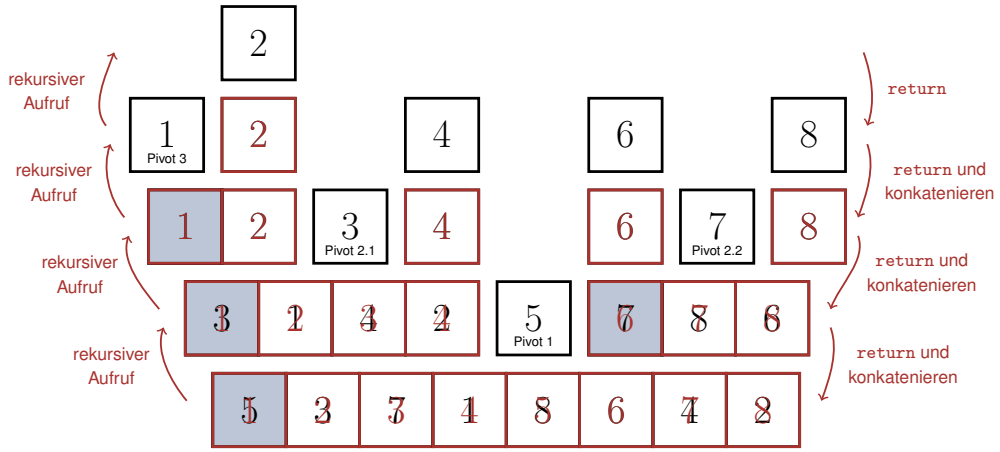
$\mathcal{O}(n \log_2 n)$-Sortierer – Quicksort

Rekursives Quicksort

- Einer der bekanntesten Sortieralgorithmen
- Im Worst-Case allerdings eine Zeitkomplexität in $\mathcal{O}(n^2)$
- Kann aber an einer gewissen Stelle **randomisiert** werden
- Erwartete Laufzeit in $\mathcal{O}(n \log_2 n)$
- Sehr gute Laufzeit in der Praxis

- Wähle beliebiges **Pivotelement** (wir nehmen hier das jeweils erste)
- Erstelle eine Liste mit kleineren und eine mit grösseren Elementen
- Rufe Algorithmus rekursiv auf diesen Listen auf
- Füge sortierte Listen jeweils zusammen

Rekursives Quicksort



Rekursives Quicksort

Quicksort als rekursive Python-Funktion

- die eine Liste `data` als Parameter übergeben erhält
- das erste Element von `data` als Pivot-Element wählt
- eine Liste mit kleineren und eine Liste mit grösseren Elementen erstellt
- den Algorithmus auf diesen rekursiv anwendet
- die zwei so sortierten Listen und das Pivot-Element konkateniert und zurückgibt

Rekursives Quicksort

```
def quicksort(data):
    if len(data) <= 1:
        return data
    else:
        pivot = data[0]
        leftdata = [i for i in data[1:] if i < pivot]
        rightdata = [i for i in data[1:] if i >= pivot]
        return quicksort(leftdata) + [pivot] + quicksort(rightdata)
```

Dictionarys

Python-Listen

- Zugriff auf Stelle `i` mit `[i]`
- Element `x` am Ende hinzufügen mit `append(x)`
- Element am Anfang oder Ende entfernen mit `pop(0)` und `pop()`
- Element an Stelle `i` hinzufügen oder entfernen mit `insert(i,x)` oder `pop(i)`
- List Comprehensions
- Test, ob Element in Liste ist mit `in`
- Iterieren über alle Elemente mit `for`-Schleife
- Teilliste erstellen von Position `i` bis `j` mit `[i:j+1]`
- ...

Einschränkung durch Zugriff per Index, wenn Daten z.B. mit diesem assoziiert sind, aber nicht für alle möglichen Indizes Daten vorliegen

Python-Dictionarys

Key-Value-Paare
Zugriff nicht über Index, sondern selbst-definierten „Schlüssel“

- Teilweise ähnliche Funktionalität wie Listen
- ... aber Daten sind nicht nach Indizes sortiert
- Initialisierung mit geschweiften Klammern

```
data = {}
```

- Schlüssel und Werte werden mit Doppelpunkt getrennt

```
data = {10 : "Wert 1", 16 : "Wert 2", 39 : "Wert 3" }
data = {"eins" : "Wert 1", "zwei" : "Wert 2", "pi" : 3.14 }
```

- Zugriff auf Wert mit Schlüssel `key` mit `[key]`

```
print(data[16])
```


Fibonacci-Zahlen

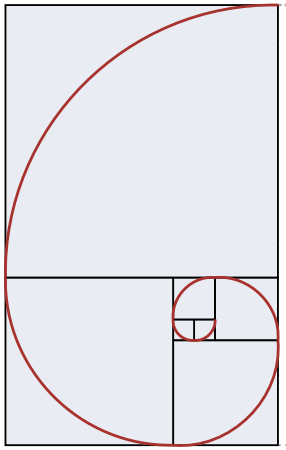
Fibonacci-Zahlen

- Die Folge der Fibonacci-Zahlen ist definiert als

$$1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots,$$

- Eine Zahl der Folge ergibt sich also aus der Summe der beiden Vorgänger; wobei die ersten beiden Zahlen 1 sind
- Sind in vielen natürlichen Phänomenen anzutreffen… oder im HB Zürich

Fibonacci-Zahlen



- Die Berechnung kann wieder iterativ oder rekursiv erfolgen
- Rekursiv definiert als

$$\text{fib}(1) = 1, \text{fib}(2) = 1$$

und

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

- Kann direkt in Python umgesetzt werden

Aufgabe – Fibonacci-Zahlen rekursiv berechnen

Implementieren Sie eine rekursive Funktion, die

- einen Parameter `n` erhält
- und die `n`-te Fibonacci-Zahl zurückgibt

Geben Sie dann die ersten 20 Fibonacci-Zahlen aus

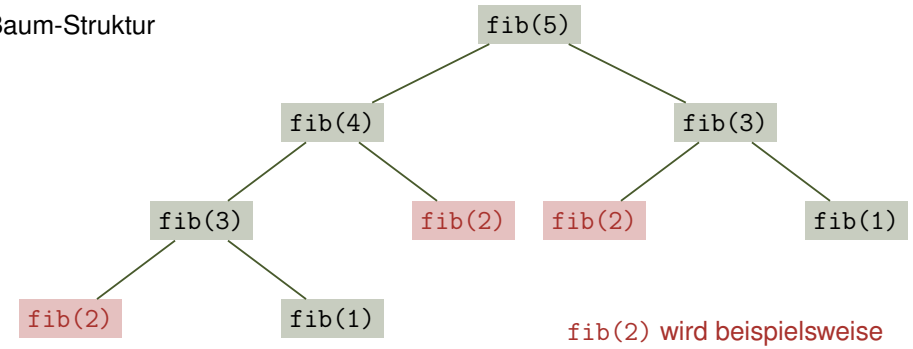
Fibonacci-Zahlen rekursiv berechnen

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

for i in range(1, 21):
    print(fib(i), end=" ")
```

Fibonacci-Zahlen rekursiv berechnen

Baum-Struktur



`fib(2)` wird beispielsweise drei Mal aufgerufen

Memoization

Memoization

- Funktion `fib` wird wiederholt mit denselben Parameterwerten aufgerufen
- Rekursion ist hier sehr viel langsamer als Iteration
- Dieses Problem gab es bei binärer Suche oder Fakultät nicht, da Aufrufe linear
- Bei Mergesort und Quicksort zwar auch Baumstruktur, aber disjunkte Aufrufe

- `fib(n)` ruft `fib(n-1)` und `fib(n-2)` auf
- `fib(n-1)` ruft wieder `fib(n-2)` auf
- `fib(n-2)` ruft beide Male wieder kompletten Teilbaum auf

Memoization

- Speichere einmal berechnete Werte, anstatt sie mehrfach zu berechnen
- Jeder Funktionsaufruf schaut zunächst, ob Wert bereits berechnet wurde
  - Falls ja, wird der Wert nicht erneut berechnet
  - Falls nein, wird der Wert berechnet und gespeichert
- Prinzip des Algorithmus bleibt sonst bestehen
- Speichere Werte in Dictionary
- Dieses kann eine globale Variable sein oder als Parameter übergeben werden

Aufgabe – Fibonacci-Zahlen mit Memoization

Implementieren Sie eine rekursive Funktion, die

- einen Parameter `n` erhält
- und die `n`-te Fibonacci-Zahl zurückgibt
- dabei ein globales Dictionary verwendet, um Memoization umzusetzen, und mit `in` nach dem entsprechenden Wert sucht

Geben Sie diesmal die ersten 200 Fibonacci-Zahlen aus

Fibonacci-Zahlen mit Memoization

```
memo = {1: 1, 2: 1}

def fib(n):
    if n in memo:
        return memo[n]
    else:
        memo[n] = fib(n-1) + fib(n-2)
        return memo[n]


for i in range(1, 201):
    print(fib(i))
```


Ähnlichkeit von DNA

Ähnlichkeit von DNA

Finde Methode zum Vergleich von DNA-Molekülen

- Suche in Genom- (oder Protein-) Datenbanken
- Erstellung von phylogenetischen Bäumen
- Teilproblem bei der DNA-Sequenzierung

- Finde geeignete Datenstruktur für die Moleküle
- Definiere geeignetes Ähnlichkeitsmass
- Entwirf einen möglichst effizienten Algorithmus zur Berechnung der Ähnlichkeit bezüglich dieses Masses

Modellierung der Daten – Moleküle als Strings

Darstellung als Strings

DNA sind lange, kettenförmige Moleküle, bestehend aus wenigen, sich oft wiederholten Grundbausteinen (Adenin, Thymin, Guanin, Cytosin)

Alignments – Ähnlichkeitsmass

Ähnlichkeitsmass soll **häufige Veränderungen** in DNA-Sequenzen widerspiegeln

- Austausch einzelner Basen oder Aminosäuren
- Einfügen oder Löschen kurzer Teilsequenzen

Alignments: Schreibe beide Strings buchstabenweise untereinander, füge dabei an beliebigen Stellen Lückensymbole ein

Eingabe: Strings $s = \texttt{GACGATTATG}$ und $t = \texttt{GATCGAATAG}$

Mögliche Alignments

$$s' = \texttt{GA–CGATTATG} \quad s'' = \texttt{GAC–GATTATG} \quad s''' = \texttt{GACGAT———TA–TG}$$
$$t' = \texttt{GATCGAATA–G} \quad t'' = \texttt{GATCGAATAG–} \quad t''' = \texttt{——GATCGAATAG——}$$

Spalten bestehend aus zwei Lücken sind sinnlos, kommen also nicht vor

Alignments – Ähnlichkeitsmass

Idee zur Bewertung

- Alignment spaltenweise bewerten, dann über alle Spalten aufsummieren
- Spalte mit Lücke erhält Kosten („penalty“) g
- Spalte mit Buchstaben a und b erhält Kosten $p(a, b)$
- $p(a, b)$ ist Null für $a = b$ und gross für $a \neq b$
- **Ziel:** Minimiere die Kosten
- Beispiel für Bewertung: **Edit-Distanz**

Edit-Distanz

Levenshtein, 1966

Zähle Mismatches und Lücken, d. h.

- $g = 1$
- $p(a, a) = 0$ und
- $p(a, b) = 1$ für $a \neq b$

Eingabe: Strings $s = \texttt{GACGATTATG}$ und $t = \texttt{GATCGAATAG}$

Mögliche Alignments

$$s' = \texttt{GA–CGATTATG} \quad s'' = \texttt{GAC–GATTATG} \quad s''' = \texttt{GACGAT———TA–TG}$$
$$t' = \texttt{GATCGAATA–G} \quad t'' = \texttt{GATCGAATAG–} \quad t''' = \texttt{——GATCGAATAG——}$$

Edit-Distanz: $d_{\text{edit}}(s', t') = 3 \quad d_{\text{edit}}(s'', t'') = 5 \quad d_{\text{edit}}(s''', t''') = 10$

Vollständige Suche

- **Frage:** Wie kann man ein optimales Alignment finden?
- **Idee:** Probiere alle möglichen Alignments durch
- **Problem:** Dies sind zu viele

Seien s und t zwei Strings der Länge n.
Dann gibt es **mehr als 3^n** mögliche Alignments von s und t.

- Alignment ist eindeutig bestimmt durch die Position der eingefügten Lücken
- **Beispiel für $n = 3$**

$$\begin{array}{c|cc|cc|c} s_1 & - & - & s_2 & s_3 \\ - & t_1 & t_2 & - & t_3 \end{array} \quad \text{oder} \quad \begin{array}{c|cc|cc|c} s_1 & s_2 & - & - & s_3 \\ t_1 & - & t_2 & t_3 & - \end{array}$$

- Bereits daraus lassen sich 3^n Alignments zusammensetzen

Exponentielle Laufzeit

n	10	50	100	300	10 000
$10n$	100	500	1 000	3 000	100 000
$3n^2$	300	7 500	30 000	270 000	300 000 000
n^3	1 000	125 000	1 000 000	27 000 000	13 Ziffern
3^n	59 049	24 Ziffern	48 Ziffern	143 Ziffern	4 772 Ziffern

- Vollständige Suche ist viel zu langsam
- **Dynamische Programmierung**

Dynamische Programmierung

Der Algorithmus von Needleman und Wunsch

Dynamische Programmierung

Lösung für gesamte Eingabe zusammensetzen aus Teillösungen für Teilprobleme, beginnend mit kleinsten Teilproblemen

- Teillösungen werden gespeichert und (wiederholt) wiederverwendet
- Verwende Tabelle
- Memoization ist eng verwandt mit dynamischer Programmierung
- Ähnlich zu Divide-and-Conquer, aber versucht, Rekursion zu vermeiden
- Bottom-Up statt Top-Down
- **Optimalitätsprinzip von Bellman:** Optimale Lösung kann aus optimalen Lösungen für Teilprobleme zusammengesetzt werden

Dynamische Programmierung

Ob DP verwendet werden kann, hängt davon ab, ob Teilprobleme definiert werden können, für die das Optimalitätsprinzip von Bellman gilt

Der wesentliche Punkt ist also, Teilprobleme geschickt zu definieren

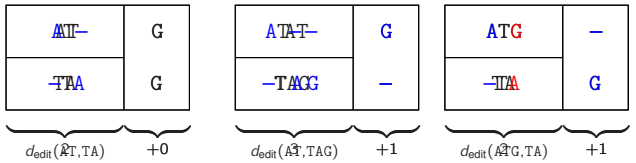
Needleman und Wunsch, 1970

- Alle Paare von Anfangsstücken (Präfixen) der gegebenen Strings als Teilprobleme
- Berechne Alignments für längere Präfixe aus den optimalen Alignments für kürzere Präfixe

Beispiel für das Alignment von Präfixen

Berechne optimales Alignment von $s = \texttt{ATG}$ und $t = \texttt{TAG}$

Unterscheide **drei Fälle** bezüglich der letzten Spalte



Optimales Alignment kann auf optimales Alignment zurückgeführt werden

Berechnung der Edit-Distanz für drei Paare von Präfixen

$t' = \texttt{–TAG}$

Initialisierung der Kosten-Tabelle

- Der **leere String λ** ist ein String der Länge 0
- Er ist Präfix von jedem anderen String

Initialisierung

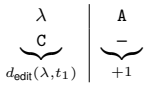
Alignment eines nichtleeren Präfixes mit λ ist eindeutig

$$\begin{matrix} s_1\, s_2 \ldots s_i \\ - - \ldots - \end{matrix} \quad \text{oder} \quad \begin{matrix} - - \ldots - \\ t_1\, t_2 \ldots t_i \end{matrix}$$

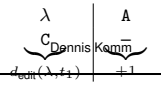
Kosten: $d_{\text{edit}}(s_1 \ldots s_i, \lambda) = d_{\text{edit}}(\lambda, t_1 \ldots t_i) = i$

Ausfüllen der Kosten-Tabelle

Initialisierung Lücke in t einfügen

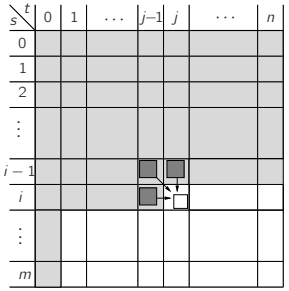


Lücke in t einfügen



Lücke in s einfügen

Ausfüllen der Kosten-Tabelle



Lücke in t — Fall 1

Lücke in s — Fall 2

Match bzw. Mismatch — Fall 3

$$d_{\text{edit}}(s_1 \dots s_i, t_1 \dots t_j) = \min\{d_{\text{edit}}(s_1 \dots s_{i-1}, t_1 \dots t_j) + 1,$$
$$d_{\text{edit}}(s_1 \dots s_i, t_1 \dots t_{j-1}) + 1,$$
$$d_{\text{edit}}(s_1 \dots s_{i-1}, t_1 \dots t_{j-1}) + p(s_i, t_j)\}$$

Dynamische Programmierung

Umsetzung in `numpy`

Alignment-Algorithmus – Initialisierung

- Verwende Modul `numpy`, das schnell mit Matrizen (Tabellen) rechnen kann

```
import numpy as np
```

- Eingabe gegeben durch zwei Strings `seq1` und `seq2`

```
seq1 = "ACTAC"
seq2 = "AACTGATGA"
m = len(seq1)
n = len(seq2)
```

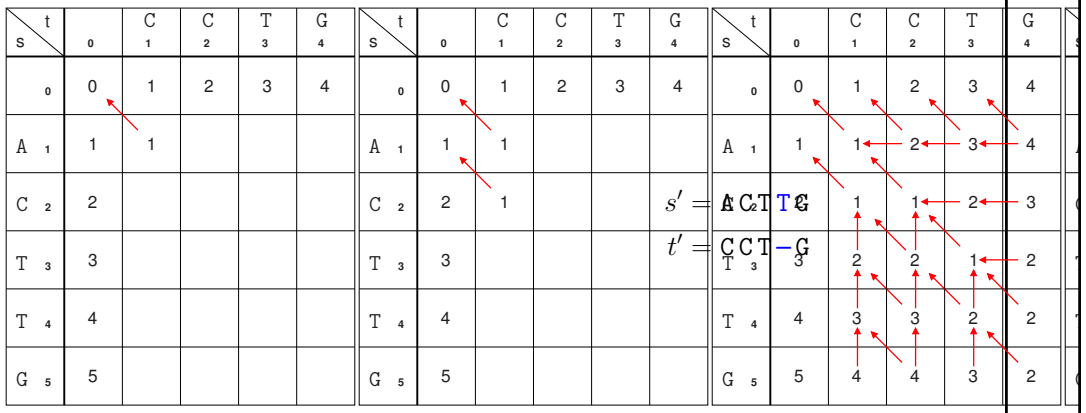
- Initialisiere Kosten-Tabelle

```
penal = np.zeros((m+1, n+1))
for j in range(0, n+1):
    penal[0][j] = j                    Erste Zeile (Alignment mit λ)
for i in range(0, m+1):
    penal[i][0] = i                    Erste Spalte (Alignment mit λ)
```


Alignment-Algorithmus – Ausfüllen der Kosten-Tabelle

```
for i in range(1, m+1):                          Für jede Zeile
  for j in range(1, n+1):                        Für jede Spalte
    if seq1[i-1] == seq2[j-1]:                   Gleiche Buchstaben in aktueller Zelle?
      pij = 0                                    Dann keine Kosten
    else:                                        Falls nicht
      pij = 1                                    Dann Kosten
    x = [penal[i-1][j] + 1, penal[i][j-1] + 1, penal[i-1][j-1] + pij]
    penal_min = np.amin(x)  Betrachte drei Möglichkeiten  Berechne Minimum dieser Möglichkeiten
    penal[i][j] = penal_min  Füge Lücke im zweiten String ein  Füge Lücke im ersten String ein  Füge Match bzw. Mismatch ein  Speichere minimalen Wert in Kosten-Tabelle
```

Bestimmung des optimalen Alignments



Alignment-Algorithmus – Initialisierung der Tracing-Tabelle

Tracing (Pfeile in der Kosten-Tabelle)

Speichere auch den Index des Minimums

Erstelle Tabelle, um Weg durch Kosten-Tabelle zu speichern...

```
penal = np.zeros((m+1, n+1))
trace = np.zeros((m+1, n+1))          Gleiche Grösse wie Kosten-Tabelle
```

```
for j in range(1, n+1):
  trace[0][j] = 1                     In erster Zeile nur Schritte von links
for i in range(1, m+1):
  trace[i][0] = 0                     In erster Spalte nur Schritte von oben
```


Alignment-Algorithmus – Ausfüllen der Tracing-Tabelle

Speichere Index des Minimums mit der `numpy`-Funktion **`argmin`**...

```
penal_min = np.amin(x)              Berechne geringste Kosten
index_min = np.argmin(x)            Berechne Index des Minimums
penal[i][j] = penal_min             Speichere Wert
trace[i][j] = index_min             Speichere Index
```

- Laufe nun rückwärts durch Tabelle `trace`
- Wert gibt Index des Minimums an
- Füge Lücke ein oder gib Match bzw. Mismatch aus
- Fahre fort in vorheriger Spalte und Zeile von `trace`
- Bei Match bzw. Mismatch ist nächster Eintrag links darüber
- `i = i-1` und `j = j-1`
- Sonst nur `i` oder `j` verkleinern

Alignment-Algorithmus – Resultat ausgeben

- Resultat des Algorithmus ist 2-dimensionale Liste **`result`** mit
  - einer Liste für den ersten String
  - und einer Liste für den zweiten String
- Beginne unten rechts in der Tabelle
- Fülle **`result`** mit dem umgedrehten Alignment
- Das Ergebnis wird danach lesbarer formatiert

```
result = [[], []]
i = m
j = n
```

Alignment-Algorithmus – Resultat ausgeben

```
while i > 0 or j > 0:                          Solange wir noch nicht oben links angekommen sind
   if trace[i][j] == 0:                                  Wenn Schritt von oben
      result[0].append(seq1[i-1])                        Dann füge Lücke im zweiten String ein
      result[1].append("-")
      i -= 1                                             Und fahre in Zeile darüber fort
   elif trace[i][j] == 1:                                Wenn Schritt von links
      result[0].append("-")                              Dann füge Lücke im ersten String ein
      result[1].append(seq2[j-1])
      j -= 1                                             Und fahre in Spalte links davon fort
   else:                                                 Wenn Schritt von oben links
      result[0].append(seq1[i-1])                        Füge Match bzw. Mismatch ein
      result[1].append(seq2[j-1])                        (Schreibe beide Buchstaben)
      i -= 1                                             Und fahre in Zelle links darüber fort
      j -= 1

for k in range(len(result[0])-1, -1, -1):
    print(result[0][k], " <-> ", result[1][k])
```


Alignment-Algorithmus – Zeitkomplexität

- Es gibt zwei Teile
  1. Ausfüllen der Tabellen
  2. Ausgabe des Resultats
- Das Ausfüllen ist aufwändiger, da jede Zelle betrachtet wird

```
for i in range(1, m+1):                                Für jede Zeile
   for j in range(1, n+1):                             Für jede Spalte
      if seq1[i-1] == seq2[j-1]                        Ein Vergleich
      ...
      x = [penal[i-1][j] + 1, penal[i][j-1] + 1, penal[i-1][j-1] + pij]
      penalty_min = amin(x)                            Zwei Vergleiche
      ...
```

$3 \cdot m \cdot n$ Vergleiche

Alignment-Algorithmus – Zeitkomplexität

Laufzeit: Ungefähr $3n^2$ für zwei Strings derselben Länge n

n	10	50	100	300	10 000
$10n$	100	500	1 000	3 000	100 000
$3n^2$	300	7 500	30 000	270 000	300 000 000
n^3	1 000	125 000	1 000 000	27 000 000	13 Ziffern
3^n	59 049	24 Ziffern	48 Ziffern	143 Ziffern	4 772 Ziffern

Vergleich zweier Gene (Grössenordnung $n \approx 10\,000$) braucht

- **100 MB** Platz und
- **weniger als 1 Minute** Zeit

Alignment-Algorithmus – Zeitkomplexität