Algorithmen und Datenstrukturen

Dynamische Programmierung:

Maximale Teilsumme ♦ RNA-Sekundärstruktur-Problem

Noch einmal
Maximale Teilsumme

Maximale Teilsumme

Name: **Maximale Teilsumme**
Eingabe: Sequenz *L* von *n* ganzen Zahlen
Ausgabe: größte Summe von Elementen einer Teilsequenz

- zwei Algorithmen:
  - Brute-Force: $\boldsymbol{O(n^3)}$
  - Teile und Herrsche: $\boldsymbol{O(n \log n)}$

Maximale Teilsumme (Brute-Force)

```
def maxTeilsumme_1(L):
    maxSumme = 0  # mindestens 0 (leere Teilsequenz)
    for i in range(0, len(L)): # untere Grenze
        for j in range(i, len(L)): # obere Grenze
            summe = 0
            for k in range(i, j+1): # Summe bilden
                summe += L[k]
            if summe > maxSumme:
                maxSumme = summe
    return maxSumme
```

Unnötige Berechnungen

- $i = 0$:
  - $j = 0$: $L[0]$
  - $j = 1$: $L[0] + L[1]$
  - $j = 2$: $L[0] + L[1] + L[2]$
  - $j = 3$: $L[0] + L[1] + L[2] + L[3]$
  - ...
- eigentlich ist die nächste Teilsumme immer die vorherige plus dem nächsten Element der Sequenz
- *Idee*: Speichern der zuvor berechneten Teilsummen in einer Tabelle (Matrix T, wobei an $T(i,j)$ die Teilsumme von $L[i]$ bis $L[j]$ gespeichert wird)

Tabellarischer Brute-Force-Algorithmus

- *Idee*: Speichern der zuvor berechneten Teilsummen in einer Tabelle (Matrix T, wobei an $T(i,j)$ die Teilsumme von $L[i]$ bis $L[j]$ gespeichert wird)

i \ j	0	1	2	...
0	$L[0]$	$L[0]+L[1]$	$L[0]+L[1]+L[2]$	...
1	0	$L[1]$	$L[1]+L[2]$	...
2	0	0	$L[2]$	...

Tabellierter Brute-Force-Algorithmus

```
def maxTeilsumme_2(L):
    maxSumme = 0  # mindestens 0 (leere Teilsequenz)
    T = []
    for i in range(0,len(L)):
        T.append([])
        for j in range(0,i):          # j < i
            T[i].append(0)
        T[i].append(L[i])             # j = i
        if T[i][i] > maxSumme: maxSumme = T[i][i]
        for j in range(i+1,len(L)): # j > i
            T[i].append(T[i][j-1] + L[j])
            if T[i][j] > maxSumme: maxSumme = T[i][j]
    return maxSumme
```

Zeit-Raum-Verschiebung

- *Bisher:* Zeitkomplexität $\boldsymbol{O(n^3)}$ und Platzbedarf für sechs Variablen (*also* $\boldsymbol{O(1)}$) (*ohne Platz für die Eingabe L*)
- *Jetzt:* nur eine Zuweisung für $T(i,j)$ mit $j \leq i$
- nur eine Addition und eine Zuweisung für $T(i,j) \rightarrow T(i,j+1)$
- neuen Wert sofort mit `maxSumme` vergleichen und `maxSumme` ggf. aktualisieren

→ **$\Theta(n^2)$ Zeit und Platz**

Verallgemeinerung

- Lösung des Problems durch Lösung von (kleineren) Teilproblemen
- Lösung aller Teilprobleme *nur einmal*, Speichern der Ergebnisse in einer Tabelle
- werden Lösungen von Teilproblemen zur Lösung eines größeren Problems benötigt: Auslesen des Tabelleneintrags

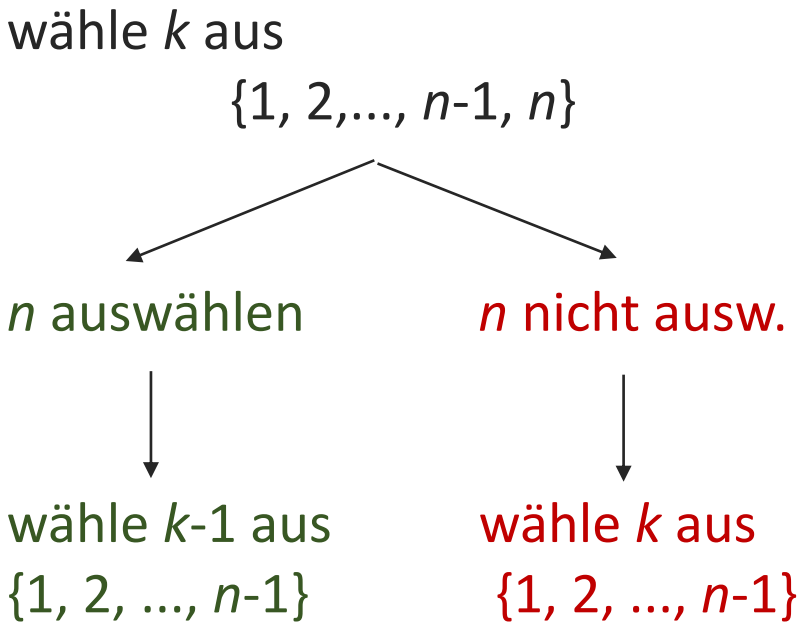
→ **Dynamische Programmierung**

- kann oft statt Rekursion verwendet werden

DP *versus* Rekursion

- *Beispiel:* Berechne Anzahl der Teilmengen mit k Elementen einer Menge mit n Elementen, also berechne $\binom{n}{k}$

- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$



DP *versus* Rekursion

- *Beispiel:* Berechne Anzahl der Teilmengen mit *k* Elementen einer Menge mit *n* Elementen, also berechne $\binom{n}{k}$

- $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

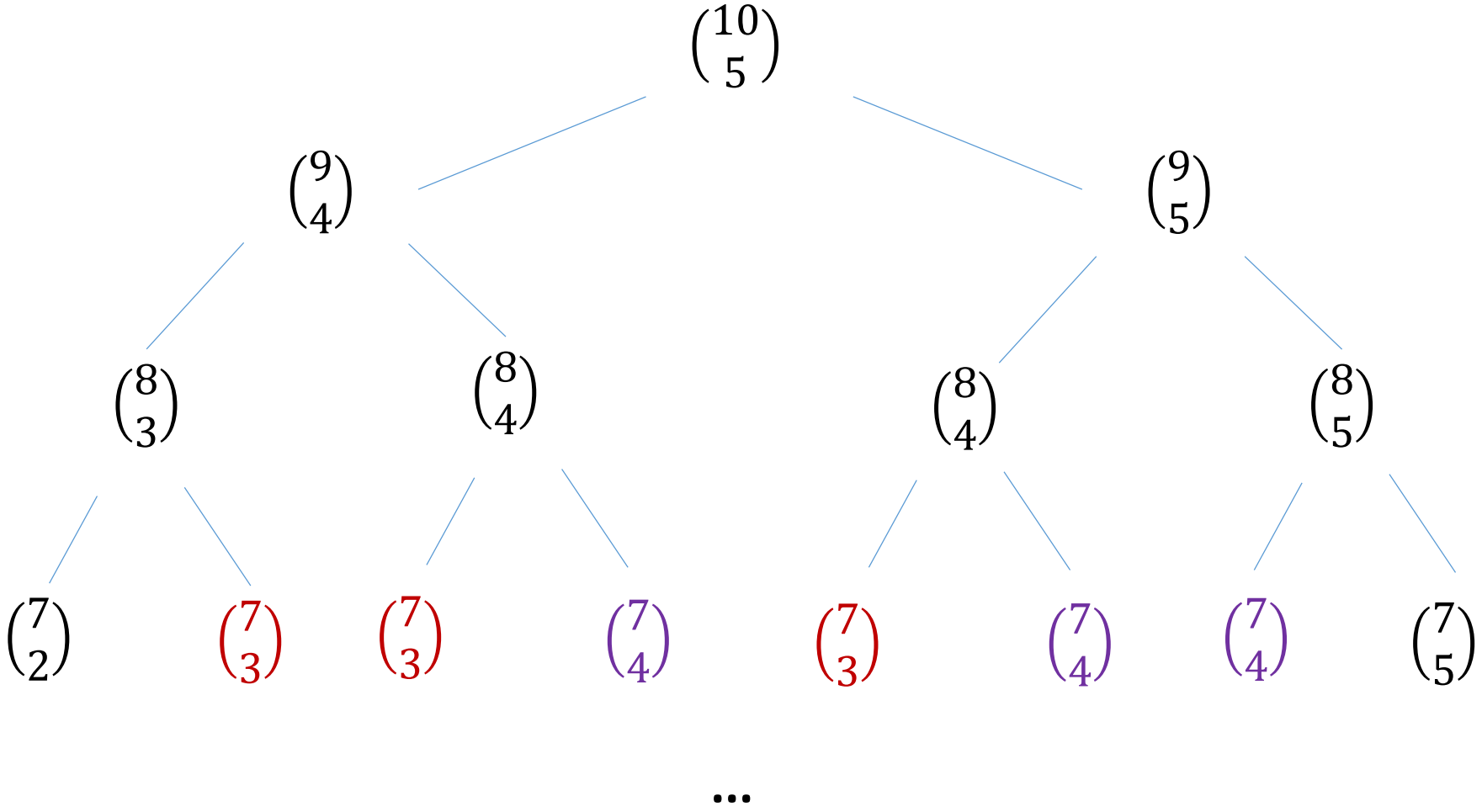
- Basisfälle:

  $\binom{n}{0} = 1$ ($n \geq 0$), $\binom{0}{k} = 0$ ($k > 0$)

Teilmengenproblem mit Rekursion

```
def compNChooseK(n,k):
        if n>=0 and k==0:
                return 1
        elif n==0 and k>0:
                return 0
        else:
                return compNChooseK(n-1,k-1)
                        + compNChooseK(n-1,k)
```

Überlappende Aufrufe bei Rekursion



Teilmengenproblem mit DP

Tabelle **Chs**

mit

$\text{Chs}(n,k) = \binom{n}{k}$

für $0 \leq k \leq n$

und

$\text{Chs}(n,k) = 0$

für $k > n$

n \ *k*	0	1	2	3	4	5
3	1	3	3	1	0	0
2	1	2	1	0	0	0
1	1	1	0	0	0	0
0	1	0	0	0	0	0

DP *versus* Rekursion

- **Rekursion:** *Top-Down-Ansatz:*
  Berechnung von $f(n)$ durch
  - Bestimmung von $m_1, m_2, ..., m_k$ kleiner als n so, dass
  - $f(n)$ einfach aus $f(m_1), f(m_2), ..., f(m_k)$ berechnet werden kann

  → z.T. müssen nicht alle Teilprobleme gelöst werden, manche dafür aber mehrfach

- **DP:** *Bottom-Up-Ansatz:*
  - Berechnung der einfachsten Teilprobleme zuerst
  - Errechnung der nächst größeren aus den bereits errechneten Lösungen, bis $f(n)$ berechnet ist

  → Berechnung aller Teilprobleme, aber jedes nur einmal

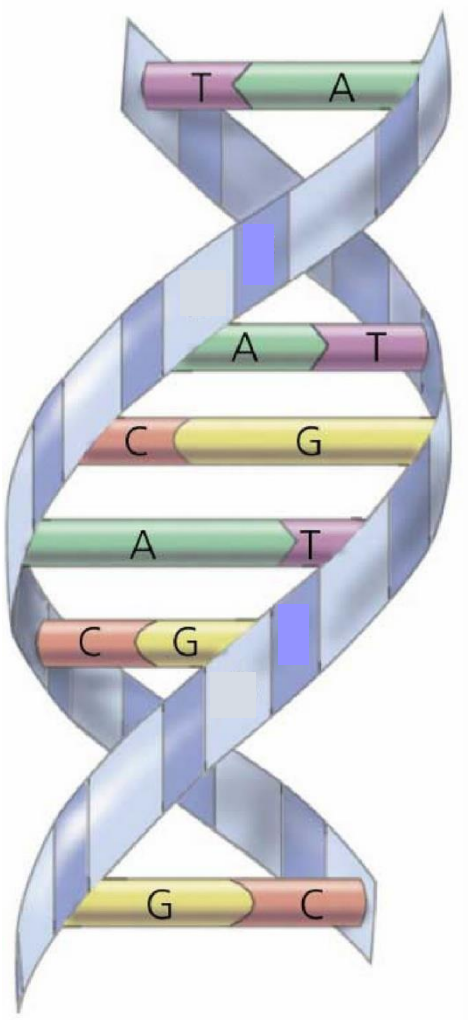
DP *versus* Rekursion (2)

- **Vorteil** von DP gegenüber Rekursion, falls viele **überlappende Teilprobleme** (*also falls viele Teilprobleme die rekursive Funktion mit denselben Parametern aufrufen*): Keine wiederholten Aufrufe der rekursiven Funktion mit denselben Parametern bei DP.

- **Nachteil** von DP: Speicherplatz für Tabellengröße muss die gesamte Laufzeit im Speicher gehalten werden.

Das RNA-Sekundärstruktur-Problem

DNA



gewundene Doppelhelix, bestehend aus:

- Adenin- (A),
- Cytosin- (C),
- Guanin- (G),
- Thymin- (T)

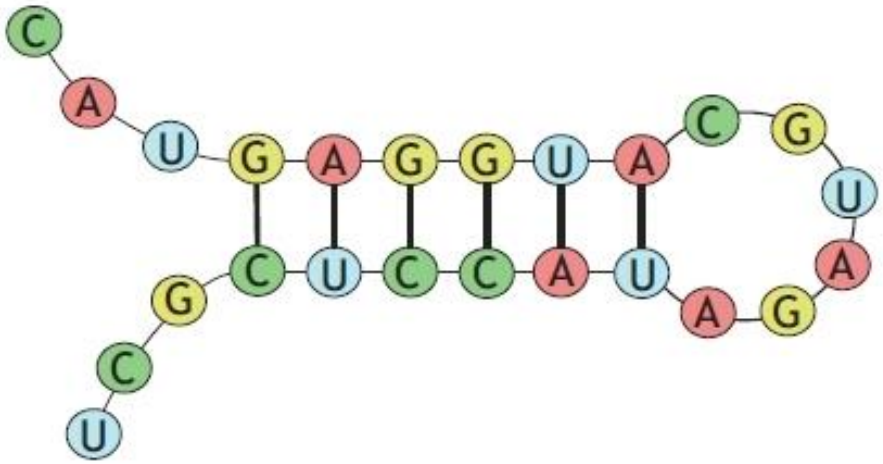
Nukleotiden

Watson-Crick-Komplementarität:
nur A und T sowie C und G können Bindungen eingehen

RNA

- Einfachstrang statt Doppelstrang
- Uracil (U) statt Thymin (T)
- verantwortlich für Proteinbildung
  - „stückweises Auftrennen“ des Doppelstrangs und
  - „Ablesen“ der DNA-Information
    mit Hilfe der Watson-Crick-Komplementarität
- besitzt **Sekundärstruktur** (Faltung)
  durch Bindung von komplementären Nukleotiden

Sekundärstruktur – Beispiel



Die stabilste Sekundärstruktur (mit minimaler freier Energie) wird angenommen, wenn eine maximale Anzahl von Nukleotid-Paaren Bindungen eingehen.

RNA-Sekundärstruktur-Problem

- Gegeben ein RNA-Strang (ohne Bindungen).
- Finde maximale Anzahl von Paaren von Nukleotiden, die eine Bindung eingehen können.
- **Eingabe:** RNA-Sequenz
- **Ausgabe:** maximale Anzahl der bindungsfähigen Nukleotiden-Paare in RNA-Sequenz

?

Algorithmisches Denken: Vom Problem zur Lösung

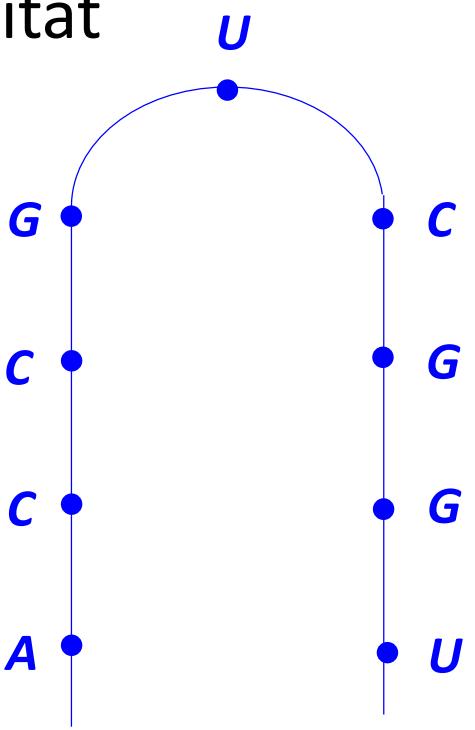
1. **Identifizieren des Problems**
2. **Formulieren des Problems**
3. **Entwurf des Algorithmus**
4. **Implementierung des Algorithmus**
5. **Anwendung des Algorithmus**

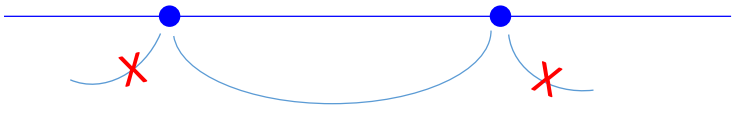
Vom Problem zum Algorithmus

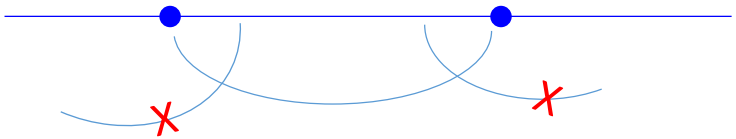
→ Problemlösung

Identifizieren des Problems: Kriterien Bindungsfähigkeit

1. Watson-Crick-Komplementarität
2. weiche Wendungen
3. Eindeutige Bindungen
4. Überlappungsfreiheit







Formulieren des Problems

- Gegeben RNA-Sequenz *w* der Länge *n*.
- **Modell** für einen RNA-Strang:
  String $w \in \{A, C, G, U\}^*$
- nuc(*i*) := Nukleotid an Position *i* $(1 \leq i \leq n)$
- wcc(*i*,*j*) $(1 \leq i < j \leq n)$ ist **True** gdw. Nukleotide an Positionen *i* und *j* sind Watson-Crick-komplementär, d.h.

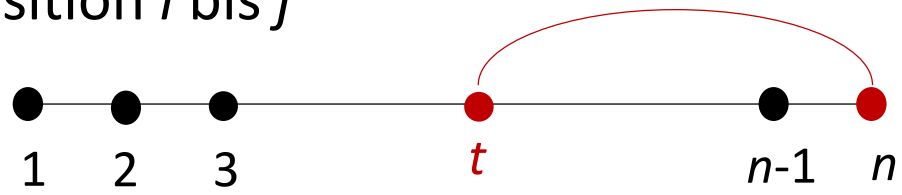
  nuc(*i*) = A $\longrightarrow$ nuc(*j*) = U
  nuc(*i*) = C $\longrightarrow$ nuc(*j*) = G
  nuc(*i*) = G $\longrightarrow$ nuc(*j*) = C
  nuc(*i*) = U $\longrightarrow$ nuc(*j*) = A

Kriterien Bindungsfähigkeit im Modell

- Gegeben RNA-Sequenz *w* der Länge *n*.
  nuc(*i*) := Nukleotid an Position *i* $(1 \leq i \leq n)$
- Ein Paar **(*i*,*j*)**, $1 \leq i < j \leq n$, ist **Matching Pair** wenn es folgende Kriterien erfüllt:
  1. Watson-Crick-Komplementarität: wcc(*i*,*j*)
  2. weiche Wendungen: $j > i + 4$
  3. Eindeutige Bindungen:
     wenn (*i*,l) oder (*k*,*j*) Matching Pair, dann $l = j$ bzw. $k = i$
  4. Überlappungsfreiheit:
     wenn (*k*,l) Matching Pair, dann
     $j < k$ oder $l < i$ oder $i < k < l < j$ oder $k < i < j < l$

Entwurf des Algorithmus

- OPT(*i*,*j*): maximale Anzahl von Matching Pairs von Position *i* bis *j*



- Suchen OPT(1,*n*)
- **1. Fall:** *n* ist keine Komponente in einem Matching Pair in maximaler Menge

  → OPT(1,*n*) = OPT(1,*n*-1)
- **2. Fall:** (*t*,*n*) ist Matching Pair in maximaler Menge

  → OPT(1,*n*) = 1 + OPT(1,*t*-1) + OPT(*t*+1,n-1)
- OPT(1,*n*) ist das Maximum der Werte aus den beiden Fällen.

Entscheidung für das Paradigma Dynamische Programmierung

- Die Option *rekursive Aufrufe* von OPT(*i*,*j*) wird zu häufigen Aufrufen der Prozedur für dieselben Parameter *i*, *j* führen
- besser: systematisch für alle Paare (*i*,*j*) aufrufen und tabellarisch die Ergebnisse speichern
- Reihenfolge der Aufrufe so, dass für die Berechnung eines neuen Wertes nur bereits berechnete Werte aus der Tabelle ausgelesen werden müssen

Pseudocode

Eingabe: $w \in \{A,C,G,U\}^*$ mit $|w| = n$

Ausgabe: OPT(1,n)

Für alle $1 \leq i \leq n$ und $0 \leq j \leq i + 4$

OPT(i,j) $\longleftarrow$ 0

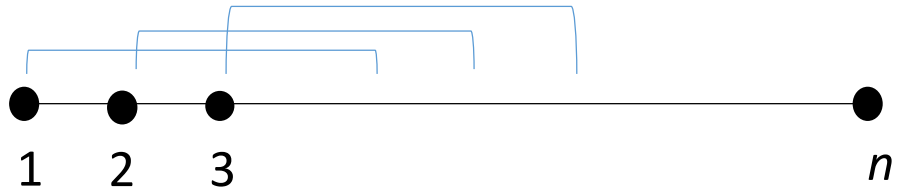
Für $5 \leq k \leq n\text{-}1$ # Distanz zwischen i und j

Für $1 \leq i \leq n\text{-}k$

$j \longleftarrow i + k$

OPT(i,j) $\longleftarrow$ max { OPT(i,j-1),

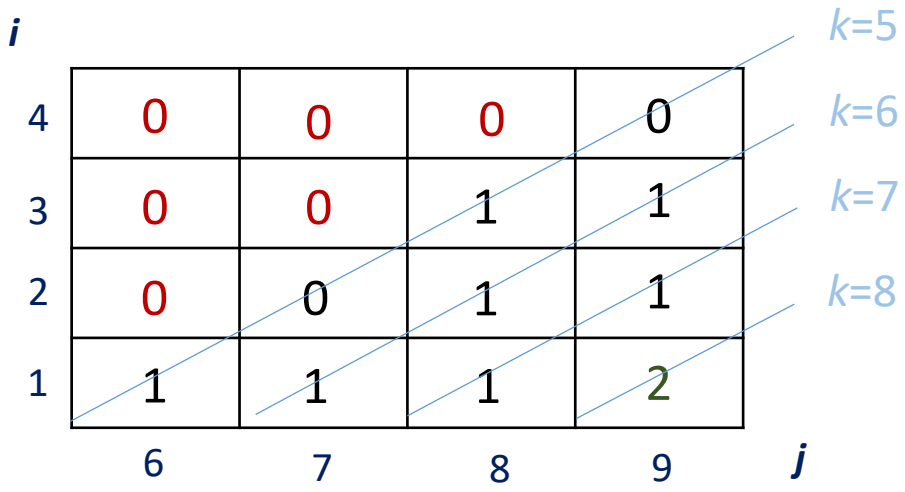
$\max_t$ { 1 + OPT(i,t-1) + OPT(t+1,j-1) | (t,j) Matching Pair }}



Gib OPT(1,n) aus

Beispiel: A C C G G U A G U

1 2 3 4 5 6 7 8 9 (n=9)



intuitiv ...

Für $5 \leq k \leq n\text{-}1$ # Distanz zwischen i und j

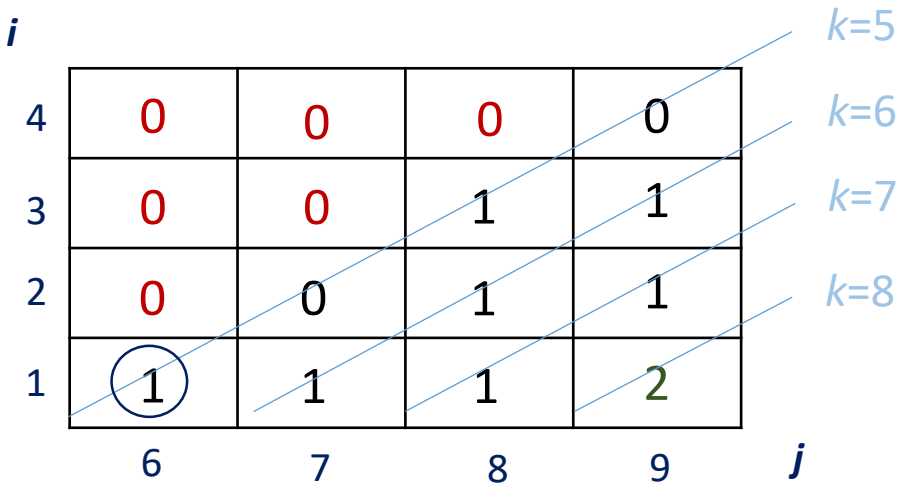
Für $1 \leq i \leq n\text{-}k$

$j \longleftarrow i + k$

$$OPT(i,j) \longleftarrow \max \{ OPT(i,j\text{-}1), \max_{t} \{ 1 + OPT(i,t\text{-}1) + OPT(t\text{+}1,j\text{-}1) \mid (t,j) \text{ Matching Pair } \}\}$$

Beispiel: A C C G G U A G U

1 2 3 4 5 6 7 8 9 (n=9)



OPT(1,6) = max { OPT(1,5),
max{1+OPT(1,0)+OPT(2,5)}}
(*t*=1)
= max { 0, 1+0+0}
= 1

Für $5 \leq k \leq n\text{-}1$ # Distanz zwischen *i* und *j*

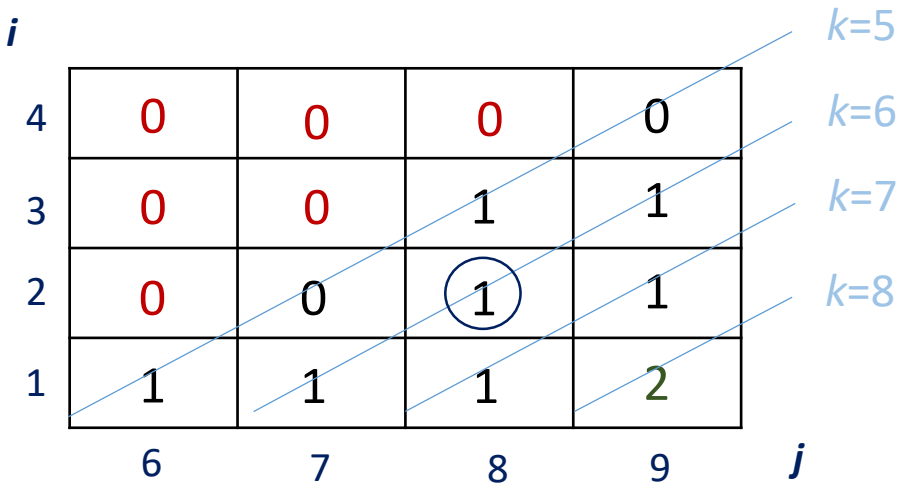
Für $1 \leq i \leq n\text{-}k$

$j \longleftarrow i + k$

$OPT(i,j) \longleftarrow \max \{ OPT(i,j\text{-}1),$
$\max_t \{ 1 + OPT(i,t\text{-}1) + OPT(t\text{+}1,j\text{-}1) \mid (t,j) \text{ Matching Pair } \}\}$

Beispiel: A C C G G U A G U

1 2 3 4 5 6 7 8 9 (n=9)



OPT(2,8) = max { OPT(2,7),
max{1+OPT(2,1)+OPT(3,7),
(*t*=2,3)
1+OPT(2,2)+OPT(4,7)}}
= max { 0, 1+0+0,
1+0+0}
= 1

Für $5 \leq k \leq n\text{-}1$ # Distanz zwischen *i* und *j*

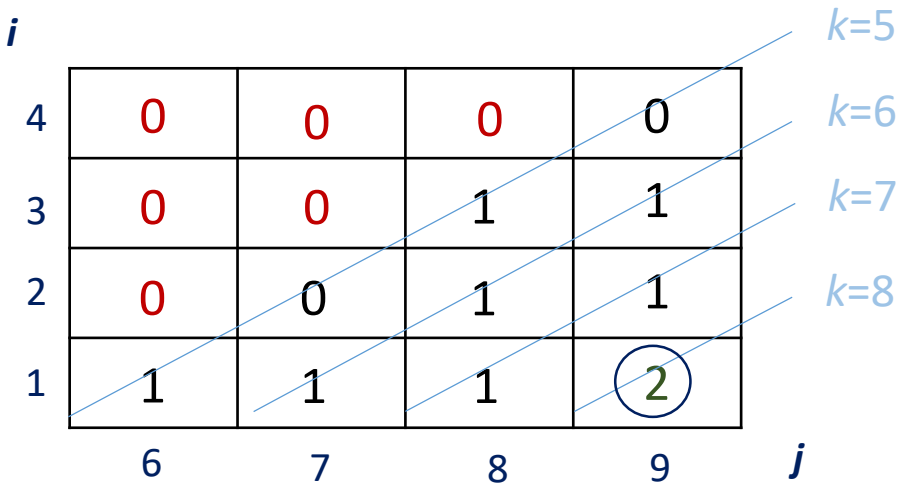
Für $1 \leq i \leq n\text{-}k$

$j \longleftarrow i + k$

$OPT(i,j) \longleftarrow \max\{ OPT(i,j\text{-}1), \max_t \{ 1 + OPT(i,t\text{-}1) + OPT(t\text{+}1,j\text{-}1) \mid (t,j) \text{ Matching Pair } \}\}$

Beispiel: A C C G G U A G U

1 2 3 4 5 6 7 8 9 (n=9)



OPT(1,9) = max { OPT(1,8),
max{1+OPT(1,0)+OPT(2,8)}}
(t=1)
= max {1, 1+0+1}
= 2

Für $5 \leq k \leq n\text{-}1$ # Distanz zwischen i und j

Für $1 \leq i \leq n\text{-}k$

$j \longleftarrow i + k$

$\text{OPT}(i,j) \longleftarrow \max \{ \text{OPT}(i,j\text{-}1),$
$\max_t \{ 1 + \text{OPT}(i,t\text{-}1) + \text{OPT}(t\text{+}1,j\text{-}1) \mid (t,j) \text{ Matching Pair } \}\}$

Zeitkomplexität

Eingabe: $w \in \{A,C,G,U\}^*$ mit $|w| = n$

Ausgabe: OPT(1,n)

$O(n^3)$

Für alle $1 \leq i \leq n$ und $0 \leq j \leq i + 4$ $O(n^2)$

  OPT(i,j) $\longleftarrow$ 0

Für $5 \leq k \leq n$-1 $O(n)$ $O(n^3)$

  Für $1 \leq i \leq n$-k $O(n)$

    $j \longleftarrow i + k$

    OPT(i,j) $\longleftarrow$ max { OPT(i,j-1),
      max { 1 + OPT(i,t-1) + OPT(t+1,j-1) | (t,j) Matching Pair }}
      $O(n)$

Gib OPT(1,n) aus

Abschließende Bemerkungen

- Finden der maximalen Menge der Matching Pairs durch „Mitschreiben“ der (t,j), die den maximalen Wert bei der Berechnung der OPT(i,j) liefern

- Für Algorithmen nach dynamischer Programmierung: Tabellengröße ist untere Schranke der Laufzeit
  - beim RNA-Sekundärstrukturproblem: Tabellengröße ist quadratisch in der Länge der RNA-Sequenz
  - wäre diese z.B. exponentiell in der Größe der Eingabe: mindestens exponentielle Laufzeit