Algorithmen auf Sequenzen
Sequenzanalyse in der Bioinformatik mit Python

Sven Rahmann

Genominformatik, Medizinische Fakultät, Universität Duisburg-Essen
und Bioinformatik für Hochdurchsatztechnologien
Informatik XI, TU Dortmund

Pycon DE, Leipzig, 6. Oktober 2011

Hintergrund

Vorlesung **Algorithmen auf Sequenzen**

Rahmen

- Informatik, TU Dortmund
- Spezialvorlesung für Diplom Informatik (auslaufend)
- Wahlvorlesung für Bachelor Informatik

Vorlesung **Algorithmen auf Sequenzen**

Rahmen

- Informatik, TU Dortmund
- Spezialvorlesung für Diplom Informatik (auslaufend)
- Wahlvorlesung für Bachelor Informatik

Inhalt

- Mustersuche (exakt, approximativ) in Texten
- Sequenzvergleiche (v.a. Bioinformatik), Alignment
- Indexdatenstrukturen (Suffixbaum, -array)

Vorlesung **Algorithmen auf Sequenzen**

Rahmen

- Informatik, TU Dortmund
- Spezialvorlesung für Diplom Informatik (auslaufend)
- Wahlvorlesung für Bachelor Informatik

Inhalt

- Mustersuche (exakt, approximativ) in Texten
- Sequenzvergleiche (v.a. Bioinformatik), Alignment
- Indexdatenstrukturen (Suffixbaum, -array)

Besonderes

- Algorithmen als Python**3**-Code angegeben

Mustersuche

Algorithmen verstehen

Wie versteht man einen Algorithmus?

Wie versteht man einen Algorithmus?

Man versteht nicht einen, sondern mehrere im Zusammenhang.

Wie versteht man einen Algorithmus?

Man versteht nicht einen, sondern mehrere im Zusammenhang.

1. Problemstellung (Eingabe, Ausgabe) verstehen
2. Selbst einen (naiven?) Algorithmus entwerfen
3. Eigenen Algorithmus auf Schwächen analysieren
4. Hauptideen verstehen
5. Beispiele betrachten

Wie versteht man einen Algorithmus?

Man versteht nicht einen, sondern mehrere im Zusammenhang.

1. Problemstellung (Eingabe, Ausgabe) verstehen
2. Selbst einen (naiven?) Algorithmus entwerfen
3. Eigenen Algorithmus auf Schwächen analysieren
4. Hauptideen verstehen
5. Beispiele betrachten
6. Mehr Beispiele betrachten!
7. Pseudocode betrachten

Wie versteht man einen Algorithmus?

Man versteht nicht einen, sondern mehrere im Zusammenhang.

1. Problemstellung (Eingabe, Ausgabe) verstehen
2. Selbst einen (naiven?) Algorithmus entwerfen
3. Eigenen Algorithmus auf Schwächen analysieren
4. Hauptideen verstehen
5. Beispiele betrachten
6. Mehr Beispiele betrachten!
7. **Python**-Code betrachten

Wie versteht man einen Algorithmus?

Man versteht nicht einen, sondern mehrere im Zusammenhang.

1. Problemstellung (Eingabe, Ausgabe) verstehen
2. Selbst einen (naiven?) Algorithmus entwerfen
3. Eigenen Algorithmus auf Schwächen analysieren
4. Hauptideen verstehen
5. Beispiele betrachten
6. Mehr Beispiele betrachten!
7. **Python**-Code betrachten

Erstes Beispiel: Mustersuche in Texten

Das Problem verstehen

Gegeben:

- Text T (über einem Alphabet)
  z.B. $T =$ mustersuchalgorithmenanalysemethode
- Muster P (über demselben Alphabet)
  z.B. $P =$ mena

Gesucht:

- Alle Intervalle $[i : j]$, so dass $T[i : j] = P$

Das Problem verstehen

Gegeben:

- Text T (über einem Alphabet)
  z.B. $T =$ mustersuchalgorithmenanalysemethode
- Muster P (über demselben Alphabet)
  z.B. $P =$ mena

Gesucht:

- Alle Intervalle $[i : j]$, so dass $T[i : j] = P$
- Oder: Alle i, so dass $T[i : i + |P|] = P$

Einen eigenen Algorithmus entwerfen

- Klar: Nur Intervalle der Länge $|P|$ müssen betrachtet werden.

Einen eigenen Algorithmus entwerfen

- Klar: Nur Intervalle der Länge $|P|$ müssen betrachtet werden.
- Idee: Beginne bei jedem i in T und vergleiche die $|P|$ folgenden Zeichen auf Übereinstimmung mit P.
  Brich bei der ersten Nichtübereinstimmung ab.

Einen eigenen Algorithmus entwerfen

- Klar: Nur Intervalle der Länge $|P|$ müssen betrachtet werden.
- Idee: Beginne bei jedem i in T und vergleiche die $|P|$ folgenden Zeichen auf Übereinstimmung mit P.
  Brich bei der ersten Nichtübereinstimmung ab.

```
1 def naive(P,T):
2     m, n = len(P), len(T)
3     for i in range(n-m+1):
4         if T[i:i+m] == P:
5             yield (i,i+m)
```

Einen eigenen Algorithmus entwerfen

- Klar: Nur Intervalle der Länge $|P|$ müssen betrachtet werden.
- Idee: Beginne bei jedem i in T und vergleiche die $|P|$ folgenden Zeichen auf Übereinstimmung mit P.
  Brich bei der ersten Nichtübereinstimmung ab.

```
1 def naive(P,T):
2     m, n = len(P), len(T)
3     for i in range(n-m+1):
4         if T[i:i+m] == P:
5             yield (i,i+m)
```

Toll: Lesbar, ausführbar, Generatorfunktion!

```
1 print(list(naive("mena","mustersuchalgo..")))
```

Schwächen des Algorithmus?

Laufzeit?

Annahme: $|P| = m \ll n = |T|$

```
1 def naive(P,T):
2     m, n = len(P), len(T)
3     for i in range(n-m+1):
4         if T[i:i+m] == P:
5             yield (i,i+m)
```

Schwächen des Algorithmus?

Laufzeit?

Annahme: $|P| = m \ll n = |T|$

```
1 def naive(P,T):
2     m, n = len(P), len(T)
3     for i in range(n-m+1):
4         if T[i:i+m] == P:
5             yield (i,i+m)
```

Im schlimmsten Fall $O(mn)$.

(Achtung: Faktor m in Zeile 4 "versteckt"!)

Schwächen des Algorithmus?

Laufzeit?
Annahme: $|P| = m \ll n = |T|$

```
1 def naive(P,T):
2     m, n = len(P), len(T)
3     for i in range(n-m+1):
4         if T[i:i+m] == P:
5             yield (i,i+m)
```

Im schlimmsten Fall $O(mn)$.
(Achtung: Faktor m in Zeile 4 "versteckt"!)

Besser wäre: Jedes Textzeichen nur 1x betrachten.

Hauptideen für einen effizienten Algorihmus

1. Jedes Textzeichen nur 1x anschauen
2. Jedes Textzeichen in konstanter Zeit verarbeiten

Hauptideen für einen effizienten Algorihmus

1. Jedes Textzeichen nur 1x anschauen
2. Jedes Textzeichen in konstanter Zeit verarbeiten
3. Speichern von Information mit Hilfe eines Automaten

Mustersuche mit Automaten

Definition (Nichtdeterministischer Automat (NFA))

Ein NFA ist ein Tupel $(Q, Q_0, F, \Sigma, \Delta)$, wobei

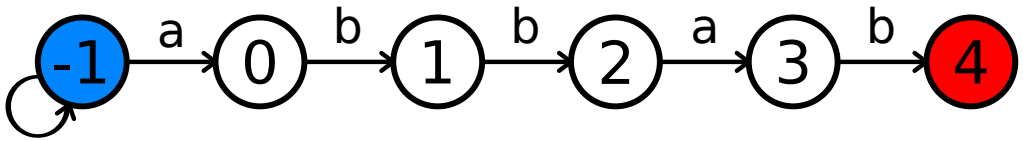
- Q: endliche Menge von Zuständen,
- $Q_0 \subset Q$: Menge von Startzuständen,
- $F \subset Q$: Menge von akzeptierenden Zuständen,
- Σ: Eingabealphabet
- $\Delta : Q \times \Sigma \rightarrow 2^Q$: nichtdeterministische Übergangsfunktion

Mustersuche mit Automaten

Definition (Nichtdeterministischer Automat (NFA))

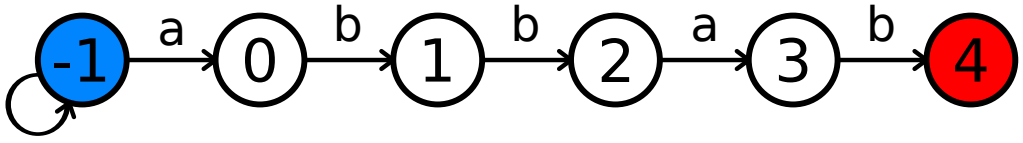
Ein NFA ist ein Tupel $(Q, Q_0, F, \Sigma, \Delta)$, wobei

- Q: endliche Menge von Zuständen,
- $Q_0 \subset Q$: Menge von Startzuständen,
- $F \subset Q$: Menge von akzeptierenden Zuständen,
- Σ: Eingabealphabet
- $\Delta : Q \times \Sigma \rightarrow 2^Q$: nichtdeterministische Übergangsfunktion



Startzustand: blau.
Akzeptierender Zustand: rot (Muster abbab gefunden).

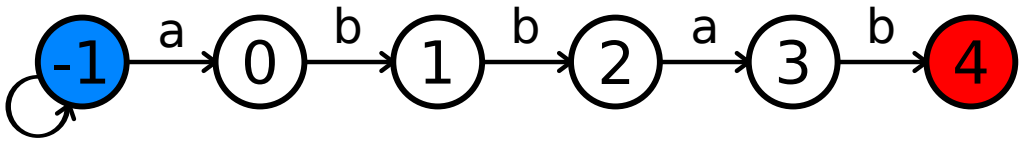
Verwaltung der Zustandsmenge



Text:

Aktiv: {-1}

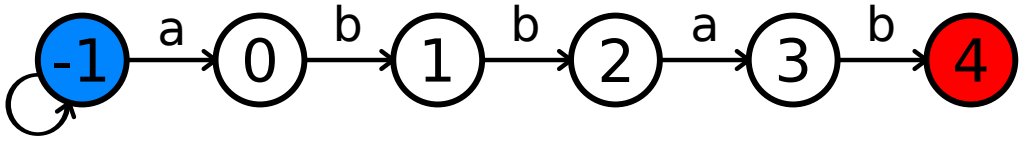
Verwaltung der Zustandsmenge



Text: b

Aktiv: {-1}{-1}

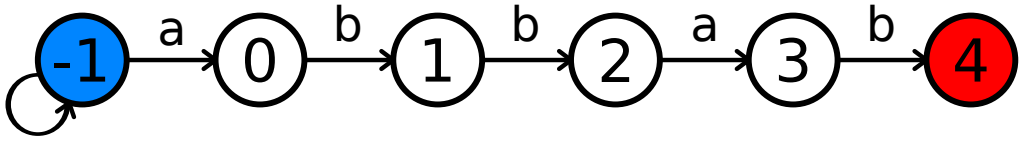
Verwaltung der Zustandsmenge



Text: b a

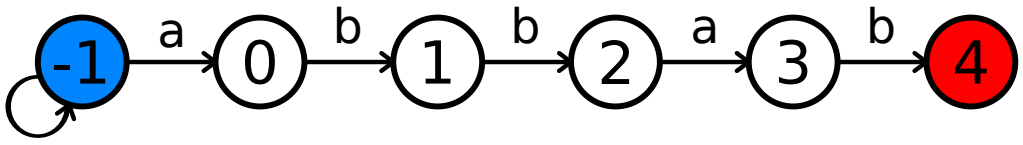
Aktiv: {-1}{-1}{-1,0}

Verwaltung der Zustandsmenge



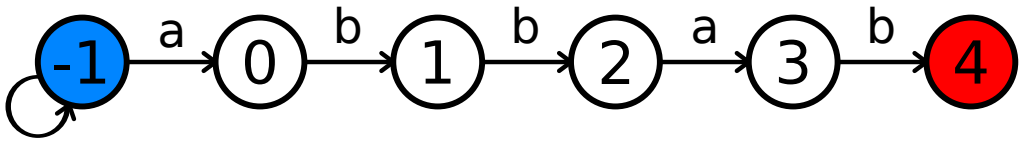
Text:		b	a	a
Aktiv:	{-1}	{-1}	{-1,0}	{-1,0}

Verwaltung der Zustandsmenge



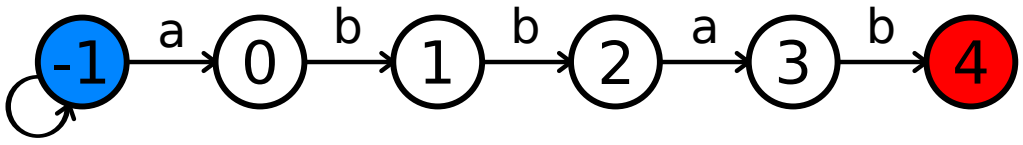
Text:		b	a	a	b
Aktiv:	{-1}	{-1}	{-1,0}	{-1,0}	{-1,1}

Verwaltung der Zustandsmenge



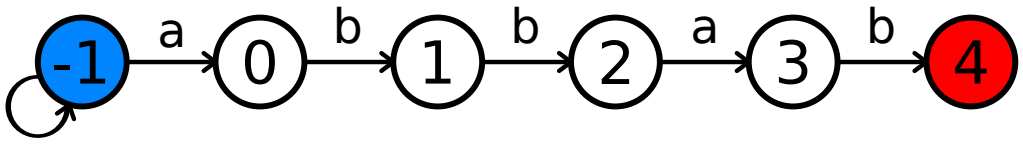
Text:		b	a	a	b	b
Aktiv:	{-1}	{-1}	{-1,0}	{-1,0}	{-1,1}	{-1,2}

Verwaltung der Zustandsmenge



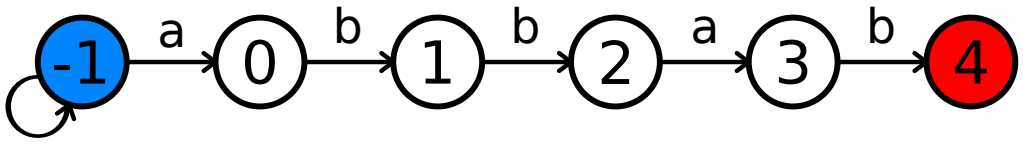
Text:		b	a	a	b	b	a
Aktiv:	{-1}	{-1}	{-1,0}	{-1,0}	{-1,1}	{-1,2}	{-1,0,3}

Verwaltung der Zustandsmenge



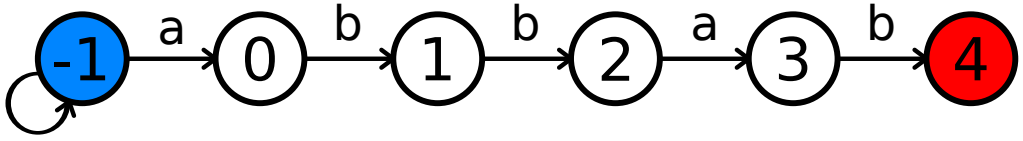
Text:		b	a	a	b	b	a	b
Aktiv:	{-1}	{-1}	{-1,0}	{-1,0}	{-1,1}	{-1,2}	{-1,0,3}	{-1,1,**4**}

Verwaltung der Zustandsmenge



Text:		b	a	a	b	b	a	b	b
Aktiv:	{-1}	{-1}	{-1,0}	{-1,0}	{-1,1}	{-1,2}	{-1,0,3}	{-1,1,**4**}	{-1,2}

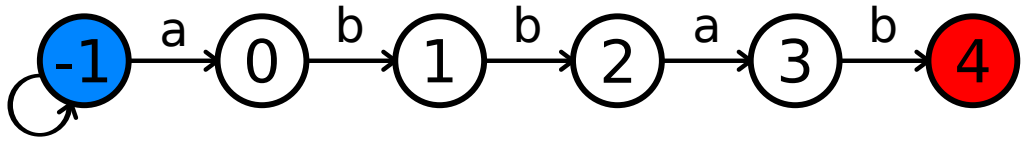
Verwaltung der Zustandsmenge



Text:		b	a	a	b	b	a	b	b
Aktiv:	{-1}	{-1}	{-1,0}	{-1,0}	{-1,1}	{-1,2}	{-1,0,3}	{-1,1,**4**}	{-1,2}

- Jedes Textzeichen wird nur 1x gelesen.
- Verwaltung der Zustandsmenge kostet $O(m)$ Zeit pro Zeichen.
- Was tun?

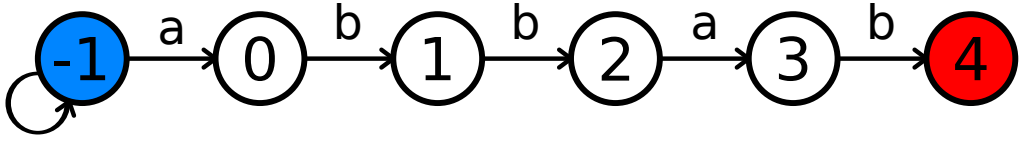
Verwaltung der Zustandsmenge



Text:		b	a	a	b	b	a	b	b
Aktiv:	{-1}	{-1}	{-1,0}	{-1,0}	{-1,1}	{-1,2}	{-1,0,3}	{-1,1,**4**}	{-1,2}

- Jedes Textzeichen wird nur 1x gelesen.
- Verwaltung der Zustandsmenge kostet $O(m)$ Zeit pro Zeichen.
- Was tun?
- Verwaltung der aktiven Zustände als Bitvektor (`int`)
- Oder: Transformation in DFA (KMP-Algorithmus)

Shift-And Algorithmus




```
1 def ShiftAnd_with_masks(T, masks, ones, accept):
2     A = 0  # bits of active states
3     for (i,c) in enumerate(T):
4         A = ((A << 1) | ones) & masks(c)
5         found = A & accept
6         if found != 0:  yield (i, found)
```

liefert Endposition eines Matches und Bitmaske;
benötigt wird aber Startposition.

Shift-And Interface und Vorbereitung

```
1 def ShiftAnd(P,T):
2     m = len(P)
3     (mask, ones, accept) = ShiftAnd_masks(P)
4     return ( (i-m+1, i+1) for (i,_)
5         in ShiftAnd_with_masks(T, mask, ones, accept)
6         )
```

Shift-And Interface und Vorbereitung

```
1 def ShiftAnd(P,T):
2     m = len(P)
3     (mask, ones, accept) = ShiftAnd_masks(P)
4     return ( (i-m+1, i+1) for (i,_)
5         in ShiftAnd_with_masks(T, mask, ones, accept)
6         )
```

```
1 def ShiftAnd_masks(P):
2     mask = dict(); bit = 1
3     for c in P:
4         if c not in mask: mask[c] = 0
5         mask[c] |= bit
6         bit *= 2
7     return (dict2function(mask,0), 1, bit//2)
```

Textindizierung

Textindizierung

Bei vielen Mustersuchen auf demselben Text T lohnt eine Indizierung von T.

Ziele dabei:

- Suche von P in T kostet nur noch $O(|P|)$ Zeit.
- Indizierung von T mit nur $O(|T|)$ Speicher und Zeit.

Textindizierung

Bei vielen Mustersuchen auf demselben Text T lohnt eine Indizierung von T.

Ziele dabei:

- Suche von P in T kostet nur noch $O(|P|)$ Zeit.
- Indizierung von T mit nur $O(|T|)$ Speicher und Zeit.

Möglichkeiten:

- Wortbasiert
- Teilstringbasiert = Suffixsortierung
  Beispiel: Suffixarray

Suffixarray eines Textes

Gegeben: Text T mit $|T| = n$.

Gesucht: Suffixarray `pos` von T,
die Permutation der Zahlen von 0 bis $n-1$,
so dass `[ T[pos[r]:] for r in range(n) ]`
die Liste der lexikographische sortierten Suffixe von T ist.

Suffixarray eines Textes

Gegeben: Text T mit $|T| = n$.

Gesucht: Suffixarray `pos` von T,
die Permutation der Zahlen von 0 bis $n-1$,
so dass `[ T[pos[r]:] for r in range(n) ]`
die Liste der lexikographische sortierten Suffixe von T ist.

Beispiel: $T =$ `cabca$`
Suffixarray `pos` $= [5, 4, 1, 2, 3, 0]$.
Sortierte Suffixe: `$, a$, abca$, bca$, ca$, cabca$`

Erstellung des Suffixarrays

Einfach dank `sort` mit `key`-Parameter:

```
1 def suffixarray(T):
2     pos = list(range(len(T)))
3     pos.sort(key = suffixes(T))
4     return pos
```

Ganz naiv in 3 Zeilen: Sortiere die Liste `[0,1,...,n-1]` nicht numerisch, sondern anhand der entsprechenden Suffixe.

Erstellung des Suffixarrays

Einfach dank `sort` mit `key`-Parameter:

```
1 def suffixarray(T):
2     pos = list(range(len(T)))
3     pos.sort(key = suffixes(T))
4     return pos
```

Ganz naiv in 3 Zeilen: Sortiere die Liste `[0,1,...,n-1]` nicht numerisch, sondern anhand der entsprechenden Suffixe.

```
1 def suffixes(T):
2     def suf(i):
3         return T[i:]
4     return suf
```

Erstellung des Suffixarrays

Einfach dank `sort` mit `key`-Parameter:

```
1 def suffixarray(T):
2     pos = list(range(len(T)))
3     pos.sort(key = suffixes(T))
4     return pos
```

Ganz naiv in 3 Zeilen: Sortiere die Liste `[0,1,...,n-1]` nicht numerisch, sondern anhand der entsprechenden Suffixe.

```
1 def suffixes(T):
2     def suf(i):
3         return T[i:]
4     return suf
```

Schwäche: Laufzeit $O(n^2 \log n)$ statt optimal $O(n)$ wegen `sort`.

Sequenzvergleich und Alignment

Edit-Distanz

Die **Edit-Distanz** von zwei Sequenzen s, t ist die kleinstmögliche Anzahl an Edit-Operationen (Ersetzung, Einfügung, Löschung eines Zeichens), mit denen sich s in t überführen lässt.

Edit-Distanz

Die **Edit-Distanz** von zwei Sequenzen s, t ist die kleinstmögliche Anzahl an Edit-Operationen (Ersetzung, Einfügung, Löschung eines Zeichens), mit denen sich s in t überführen lässt.

Einfache Fälle:

$$
\begin{aligned}
d(s, \varepsilon) &= |s|, \\
d(\varepsilon, t) &= |t|, \\
d(a, b) &= \begin{cases} 1 & \text{falls } a \neq b, \\ 0 & \text{falls } a = b. \end{cases}
\end{aligned}
$$

Allgemeiner Fall

Berechnung über Präfixe von s und t:

s	a
t	b

s	a
tb	-

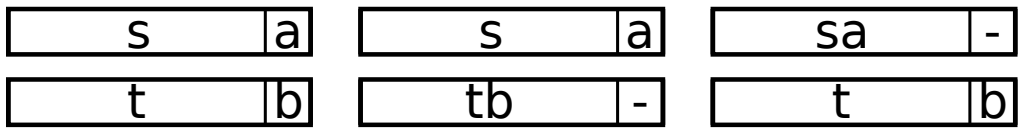
sa	-
t	b

$$d(sa, tb) = \min \left\{ \begin{array}{l} d(s,t) + d(a,b), \\ d(s,tb) + 1, \\ d(sa,t) + 1 \end{array} \right\}$$

Minimierung über die Lösung von 3 kleineren Unterproblemen.
Dynamische Programmierung statt Rekursion!

Allgemeiner Fall

Berechnung über Präfixe von s und t:



$$d(sa, tb) = \min \left\{ \begin{array}{l} d(s,t) + d(a,b), \\ d(s,tb) + 1, \\ d(sa,t) + 1 \end{array} \right\}$$

Minimierung über die Lösung von 3 kleineren Unterproblemen.
Dynamische Programmierung statt Rekursion!

Alternativ mit Scores statt Distanzen/Kosten: Maximierung.
Traceback: Rekonstruktion der Edit-Operationen.

Implementierung mit Dynamic Programming

```
1  def align_global(x,y, score=standardscore, gap=-1):
2      m, n = len(x), len(y)
3      So = [None]*(n+1)  # alte Zeile
4      Si = [j*gap for j in range(n+1)]  # akt. Zeile
5      T  = [[(0,0)]+[(0,j) for j in range(n)]]
6      for i in range(1,m+1):
7          So,Si = Si,So
8          Si[0] = i*gap
9          T.append([(i-1,0)]+[(None,None)]*n)
10         for j in range(1,n+1):
11             Si[j], T[i][j] = max(
12                 (So[j-1]+score(x[i-1],y[j-1]), (i-1,j
13                 (Si[j-1]+gap, (i,j-1)),
14                 (So[j]+gap,   (i-1,j))   )
15     return(Si[n], traceback(T,m,n,x,y))
```

Zum Mitnehmen: Vorteile von Python in algorithmischen Vorlesungen

Eingebaute Typen (Tupel, Mengen, Counter, etc.)

- Verwende eingebaute Typen wenn möglich,
  z.B. Matches als Tupel, nicht als Objeke eigener Klassen.
- Vorteil 1: Code bleibt kurz.
- Vorteil 2: Code bleibt nahe an formaler Definition.
- Aber: Kenne die Grenzen!

Eingebaute Typen (Tupel, Mengen, Counter, etc.)

- Verwende eingebaute Typen wenn möglich,
  z.B. Matches als Tupel, nicht als Objeke eigener Klassen.
- Vorteil 1: Code bleibt kurz.
- Vorteil 2: Code bleibt nahe an formaler Definition.
- Aber: Kenne die Grenzen!

Definition

Ein Hidden Markov Model ist ein Tupel 5-Tupel (Q, q_0, a, E, e), wobei ...

Eingebaute Typen (Tupel, Mengen, Counter, etc.)

- Verwende eingebaute Typen wenn möglich,
  z.B. Matches als Tupel, nicht als Objeke eigener Klassen.
- Vorteil 1: Code bleibt kurz.
- Vorteil 2: Code bleibt nahe an formaler Definition.
- Aber: Kenne die Grenzen!

Definition

Ein Hidden Markov Model ist ein Tupel 5-Tupel (Q, q_0, a, E, e), wobei ...

Vielleicht doch lieber eine Klasse `HMM` ?

Generatorfunktionen

- Verwende Generatorfunktionen wenn möglich, z.B. Matches von P in T nacheinander **yield**en.
- Vorteil 1: kein Speicherplatz für lange Listen nötig
- Vorteil 2: Benutzer entscheidet über Container (list, tuple, set)

Vorteile von Python...

...zur Darstellung von Algorithmen:

- kurze Funktionen, kein begin/end, keine { }
- häufig direkt mathematische Notation verwendbar
- Trick: `max` über Tupel (?)
- Tupel: mehrteilige Rückgabewerte, ohne Klassen zu definieren
- Generatorfunktionen: speichereffizient