

Algorithmen 1

Dynamische Programmierung



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(*Number n*)

$F :=$ Array of size $n + 1$

$F[1] := 1$

$F[2] := 1$

for $i \in \{3, \dots, n\}$ **do**

$F[i] := F[i - 1] + F[i - 2]$

return $F[n]$

F(*Number n*)

if $n = 1$ **or** $n = 2$ **then**

return 1

return $F(n - 1) + F(n - 2)$

Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  | F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

F(Number n)

```

if n = 1 or n = 2 then
  | return 1
return F(n - 1) + F(n - 2)
  
```

Laufzeit

- iterativ: $\Theta(n)$

Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  | F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

F(Number n)

```

if n = 1 or n = 2 then
  | return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:

F(6)

Laufzeit

- iterativ: $\Theta(n)$

Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

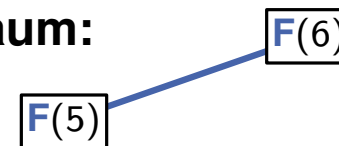
F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  | F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

F(Number n)

```

if n = 1 or n = 2 then
  | return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Laufzeit

- iterativ: $\Theta(n)$

Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

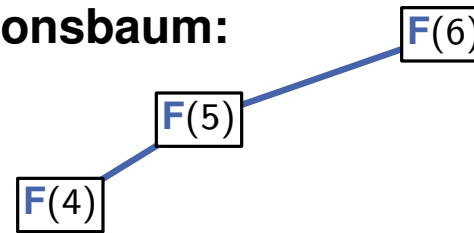
F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Laufzeit

- iterativ: $\Theta(n)$

Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  | F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

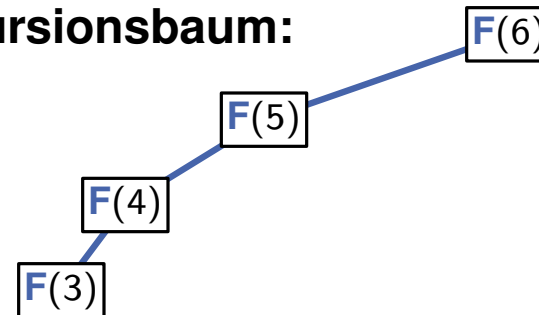
- iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  | return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

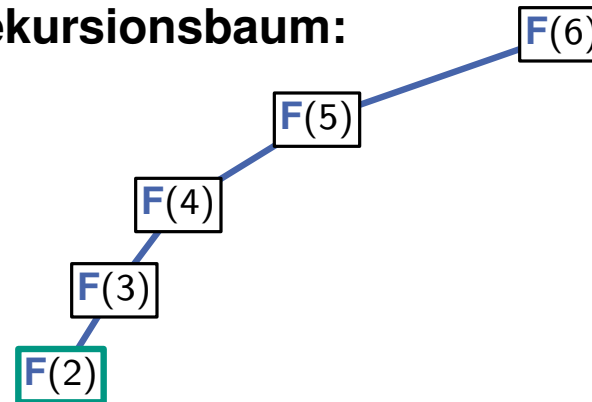
■ iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

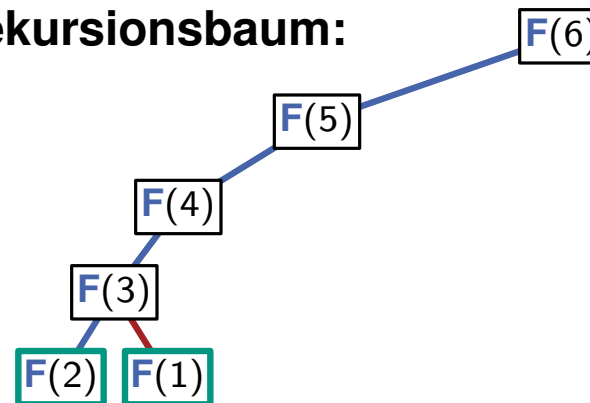
■ iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  | F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

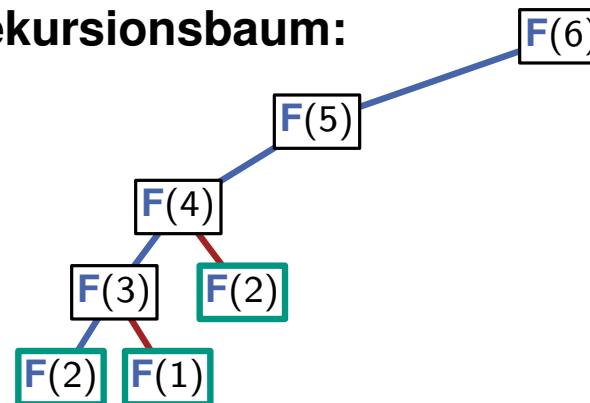
- iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  | return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

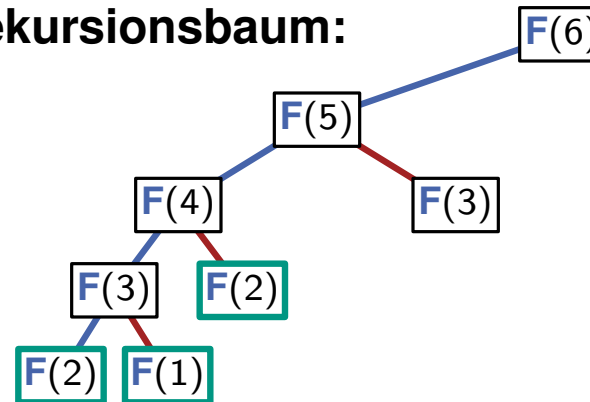
■ iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

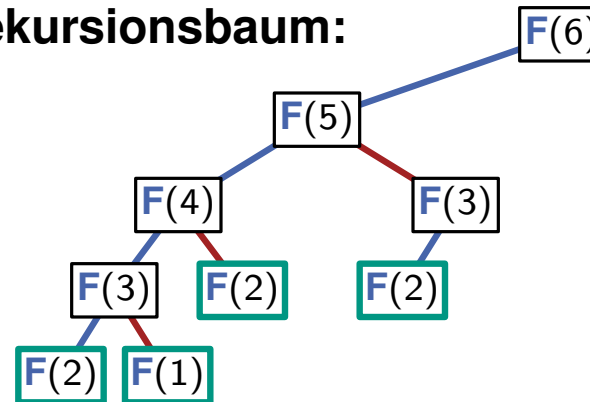
■ iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

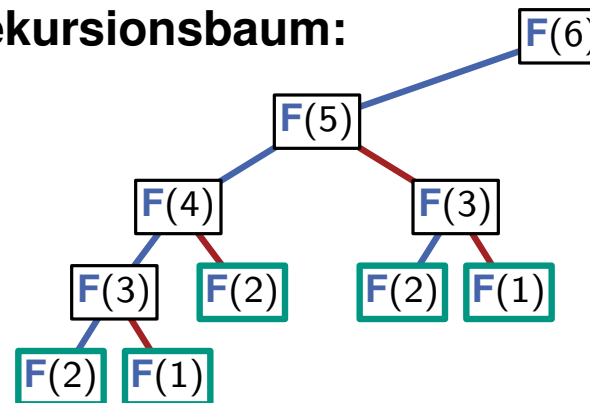
- iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```
F := Array of size n + 1
```

```
F[1] := 1
```

```
F[2] := 1
```

```
for i ∈ {3, ..., n} do
```

```
  F[i] := F[i - 1] + F[i - 2]
```

```
return F[n]
```

Laufzeit

■ iterativ: $\Theta(n)$

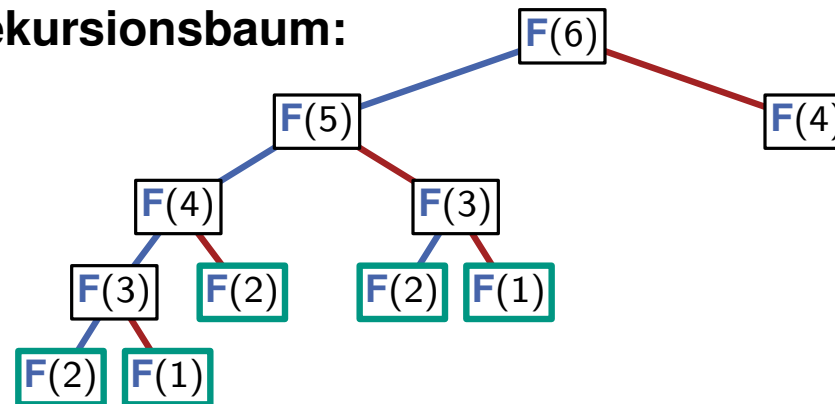
F(Number n)

```
if n = 1 or n = 2 then
```

```
  return 1
```

```
return F(n - 1) + F(n - 2)
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```
 $F :=$  Array of size  $n + 1$ 
```

```
 $F[1] := 1$ 
```

```
 $F[2] := 1$ 
```

```
for  $i \in \{3, \dots, n\}$  do
```

```
   $F[i] := F[i - 1] + F[i - 2]$ 
```

```
return  $F[n]$ 
```

Laufzeit

■ iterativ: $\Theta(n)$

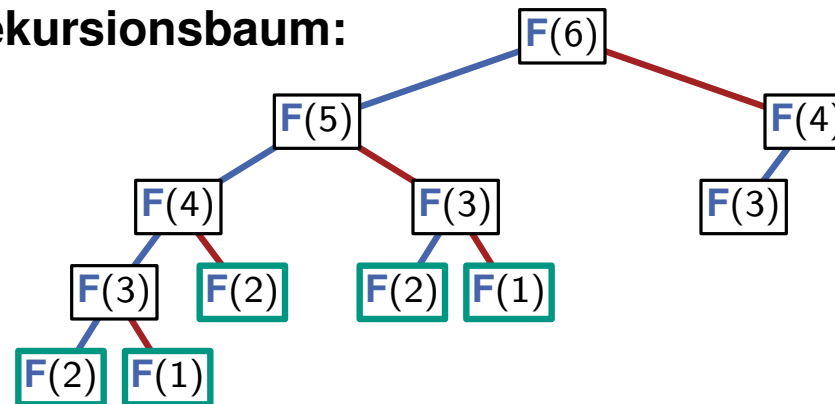
F(Number n)

```
if  $n = 1$  or  $n = 2$  then
```

```
  return 1
```

```
return  $F(n - 1) + F(n - 2)$ 
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

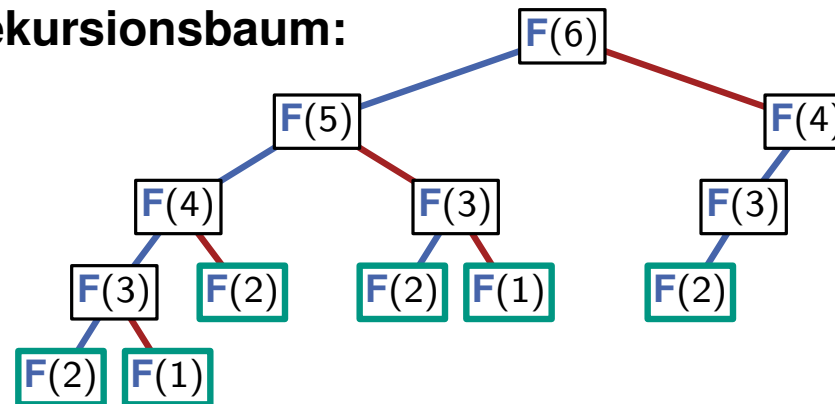
- iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

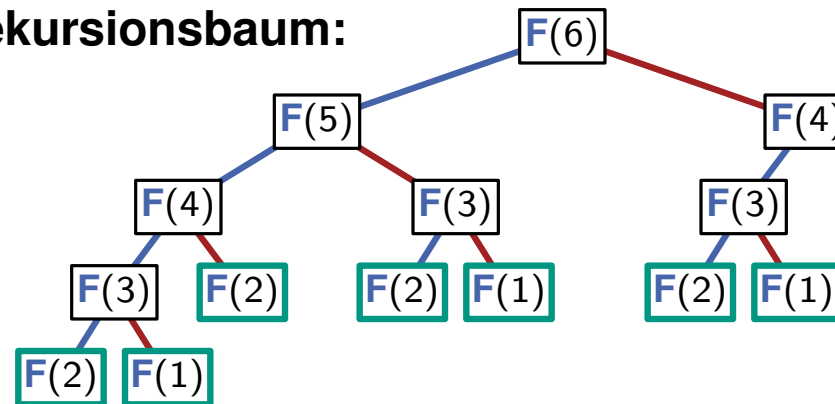
■ iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

```

F := Array of size n + 1
F[1] := 1
F[2] := 1
for i ∈ {3, ..., n} do
  F[i] := F[i - 1] + F[i - 2]
return F[n]
  
```

Laufzeit

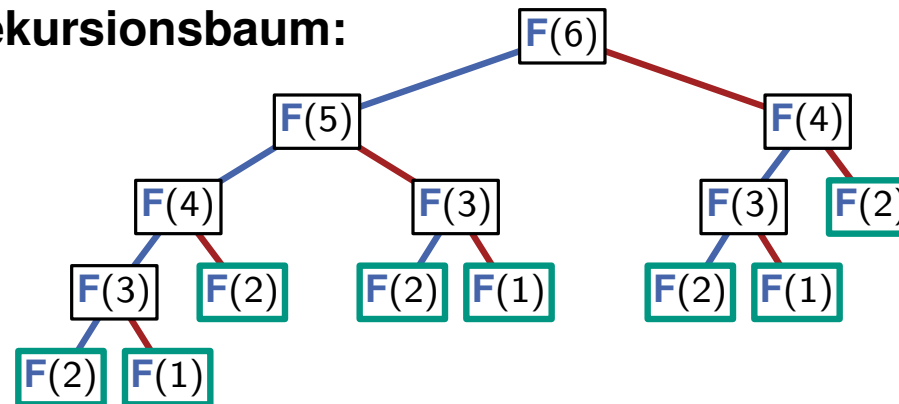
■ iterativ: $\Theta(n)$

F(Number n)

```

if n = 1 or n = 2 then
  return 1
return F(n - 1) + F(n - 2)
  
```

Rekursionsbaum:



Fibonacci-Zahlen

Definition: $F(1) = F(2) = 1$, $F(n) = F(n - 1) + F(n - 2)$ für $n \geq 3$ (1, 1, 2, 3, 5, 8, 13, 21, ...)

Fib(Number n)

$F :=$ Array of size $n + 1$

$F[1] := 1$

$F[2] := 1$

for $i \in \{3, \dots, n\}$ **do**

$F[i] := F[i - 1] + F[i - 2]$

return $F[n]$

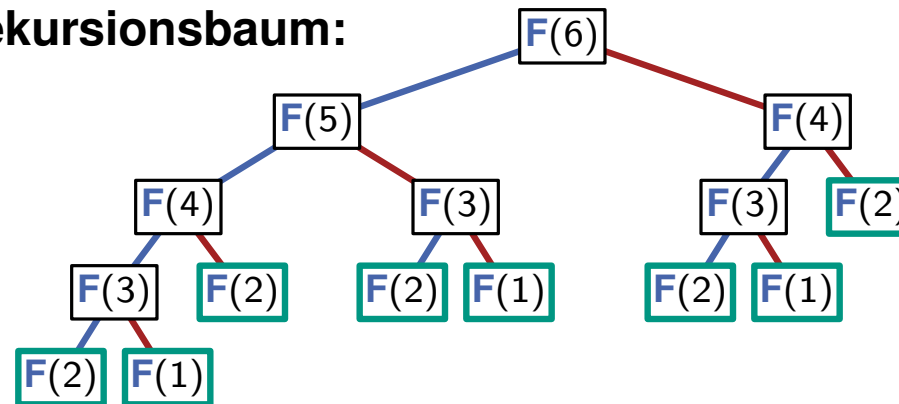
F(Number n)

if $n = 1$ **or** $n = 2$ **then**

return 1

return $F(n - 1) + F(n - 2)$

Rekursionsbaum:



Laufzeit

■ iterativ: $\Theta(n)$

■ rekursiv: $\Theta(\varphi^n)$ mit $\varphi = \frac{1+\sqrt{5}}{2} \approx 1,6181$

(Rekurrenz lösen (nicht hier): $T(n) = T(n - 1) + T(n - 2) + \Theta(1)$)

Was ist hier passiert?

Rekursiv definiertes Problem

- reduziert Gesamtproblem auf kleinere Teilprobleme
 - Gesamtproblem: berechne $F(n)$; Teilprobleme: berechne $F(n - 1)$ und $F(n - 2)$
- Teilprobleme überlappen sich
 - für $F(n - 1)$ muss man nochmal $F(n - 2)$ berechnen
- rekursive Lösung: betrachtet selbes Teilproblem mehrfach

Was ist hier passiert?

Rekursiv definiertes Problem

- reduziert Gesamtproblem auf kleinere Teilprobleme
 - Gesamtproblem: berechne $F(n)$; Teilprobleme: berechne $F(n - 1)$ und $F(n - 2)$
- Teilprobleme überlappen sich
 - für $F(n - 1)$ muss man nochmal $F(n - 2)$ berechnen
- rekursive Lösung: betrachtet selbes Teilproblem mehrfach

Iterative Lösung

- baue schrittweise größere Teillösungen auf
- Teillösungen speichern → keine mehrfache Berechnung

Was ist hier passiert?

Rekursiv definiertes Problem

- reduziert Gesamtproblem auf kleinere Teilprobleme
 - Gesamtproblem: berechne $F(n)$; Teilprobleme: berechne $F(n - 1)$ und $F(n - 2)$
- Teilprobleme überlappen sich
 - für $F(n - 1)$ muss man nochmal $F(n - 2)$ berechnen
- rekursive Lösung: betrachtet selbes Teilproblem mehrfach

Iterative Lösung

- baue schrittweise größere Teillösungen auf
- Teillösungen speichern → keine mehrfache Berechnung

Und wenn das Problem gar nicht rekursiv definiert ist?

- finde eine äquivalente rekursive Definition

Was ist hier passiert?

Rekursiv definiertes Problem

- reduziert Gesamtproblem auf kleinere Teilprobleme
 - Gesamtproblem: berechne $F(n)$; Teilprobleme: berechne $F(n - 1)$ und $F(n - 2)$
- Teilprobleme überlappen sich
 - für $F(n - 1)$ muss man nochmal $F(n - 2)$ berechnen
- rekursive Lösung: betrachtet selbes Teilproblem mehrfach

Iterative Lösung

- baue schrittweise größere Teillösungen auf
 - Teillösungen speichern → keine mehrfache Berechnung
- } meist recht einfach

Und wenn das Problem gar nicht rekursiv definiert ist?

- finde eine äquivalente rekursive Definition
- } das ist der schwierige Teil

Problemstellung: Splittability

Definition

Ein **String** S der Länge n ist eine Folge von Buchstaben $\langle S[0], \dots, S[n-1] \rangle$. Die Teilfolge $\langle S[a], \dots, S[b] \rangle$ bezeichnen wir auch mit $S[a, b] = S[a, b+1) = S(a-1, b] = S(a-1, b+1)$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
E	i	n	B	e	i	s	p	i	e	l	S	t	r	i	n	g

$S[3, 11)$

Problemstellung: Splittability

Definition

Ein **String** S der Länge n ist eine Folge von Buchstaben $\langle S[0], \dots, S[n-1] \rangle$. Die Teilfolge $\langle S[a], \dots, S[b] \rangle$ bezeichnen wir auch mit $S[a, b] = S[a, b+1) = S(a-1, b] = S(a-1, b+1)$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
E	i	n	B	e	i	s	p	i	e	l	S	t	r	i	n	g

$S[3, 11)$

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

m	o	n	k	e	y	e	a	r	t	h	a	i	r
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Problemstellung: Splittability

Definition

Ein **String** S der Länge n ist eine Folge von Buchstaben $\langle S[0], \dots, S[n-1] \rangle$. Die Teilfolge $\langle S[a], \dots, S[b] \rangle$ bezeichnen wir auch mit $S[a, b] = S[a, b+1) = S(a-1, b) = S(a-1, b+1)$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
E	i	n	B	e	i	s	p	i	e	l	S	t	r	i	n	g

$S[3, 11)$

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

m	o	n	k	e	y	e	a	r	t	h	a	i	r
↑					↑						↑	↑	
s_0					s_1						s_2	s_3	

Problemstellung: Splittability

Definition

Ein **String** S der Länge n ist eine Folge von Buchstaben $\langle S[0], \dots, S[n-1] \rangle$. Die Teilfolge $\langle S[a], \dots, S[b] \rangle$ bezeichnen wir auch mit $S[a, b] = S[a, b+1) = S(a-1, b] = S(a-1, b+1)$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
E	i	n	B	e	i	s	p	i	e	l	S	t	r	i	n	g

$S[3, 11)$

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

m	o	n	k	e	y	e	a	r	t	h	a	i	r
↑				↑			↑		↑			↑	
s_0				s_1			s_2		s_3			s_4	

Teilprobleme und rekursive Definition

Problem: **Splittability**

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

Teilprobleme und rekursive Definition

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

Teilprobleme

- für $i \in [0, n]$: Ist $S[0, i)$ splittable?

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel: $S[0, 9)$ ist splittable

Teilprobleme und rekursive Definition

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

Teilprobleme

- für $i \in [0, n]$: Ist $S[0, i)$ splittable?
- beachte: Gesamtproblem fragt ob $S[0, n)$ splittable ist

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel: $S[0, 9)$ ist splittable

Teilprobleme und rekursive Definition

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

Teilprobleme

- für $i \in [0, n]$: Ist $S[0, i)$ splittable?
- beachte: Gesamtproblem fragt ob $S[0, n)$ splittable ist

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel: $S[0, 9)$ ist splittable

Rekursive Definition auf Basis der Teilprobleme

- **splittable**(0) = true

Teilprobleme und rekursive Definition

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

Teilprobleme

- für $i \in [0, n]$: Ist $S[0, i)$ splittable?
- beachte: Gesamtproblem fragt ob $S[0, n)$ splittable ist

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel: $S[0, 9)$ ist splittable

Rekursive Definition auf Basis der Teilprobleme

- **splittable**(0) = true

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel:

splittable(7) \wedge **isWord**($S[7, 10)$) = true
 \Rightarrow **splittable**(10) = true



Teilprobleme und rekursive Definition

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

Teilprobleme

- für $i \in [0, n]$: Ist $S[0, i)$ splittable?
- beachte: Gesamtproblem fragt ob $S[0, n)$ splittable ist

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel: $S[0, 9)$ ist splittable

Rekursive Definition auf Basis der Teilprobleme

- **splittable**(0) = true

Wie geht die rekursive Definition weiter?

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel:

splittable(7) \wedge **isWord**($S[7, 10)$) = true
 \Rightarrow **splittable**(10) = true

Teilprobleme und rekursive Definition

Problem: Splittability

Gegeben ein String S , gibt es Trennstellen $0 = s_0 < \dots < s_k = n$, sodass $S[s_{i-1}, s_i)$ für alle $i \in [1, k]$ ein gültiges Wort ist? Wir sagen dann, dass S **splittable** ist.

Teilprobleme

- für $i \in [0, n]$: Ist $S[0, i)$ splittable?
- beachte: Gesamtproblem fragt ob $S[0, n)$ splittable ist

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel: $S[0, 9)$ ist splittable

Rekursive Definition auf Basis der Teilprobleme

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i)$)

0	1	2	3	4	5	6	7	8	9	10	11	12	13
m	o	n	k	e	y	e	a	r	t	h	a	i	r

Beispiel:

splittable(7) \wedge **isWord**($S[7, 10)$) = true
 \Rightarrow **splittable**(10) = true

Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn $\text{splittable}(i) = \text{true}$.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und $\text{splittable}(0) = \text{true}$

Erinnerung: Rekursive Definition von **splittable**

- $\text{splittable}(0) = \text{true}$
- $\text{splittable}(i) = \exists j \in [0, i) \text{ splittable}(j) \wedge \text{isWord}(S[j, i))$

Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splittable**(i) = true.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splittable**(0) = true
- Hinrichtung: $S[0, i)$ splittable \Rightarrow **splittable**(i) = true

Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i)$)

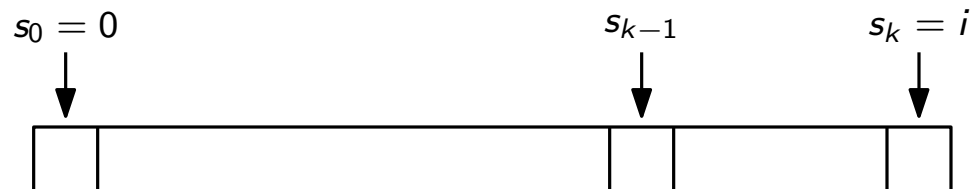
Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splittable**(i) = true.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splittable**(0) = true
- Hinrichtung: $S[0, i)$ splittable \Rightarrow **splittable**(i) = true
 - betrachte Lösung $0 = s_0 < \dots < s_k = i$



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i)$)

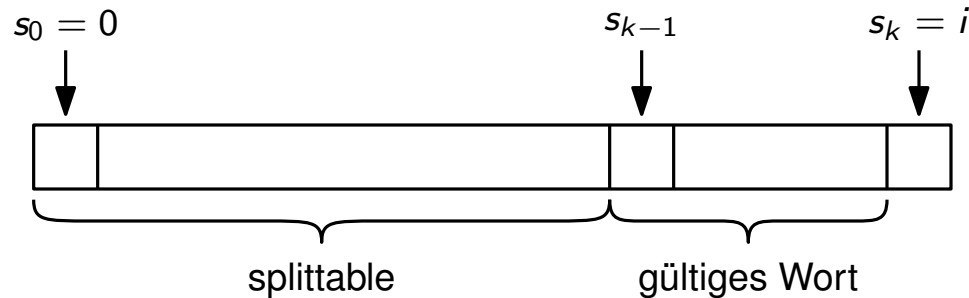
Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splittable**(i) = true.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splittable**(0) = true
- Hinrichtung: $S[0, i)$ splittable \Rightarrow **splittable**(i) = true
 - betrachte Lösung $0 = s_0 < \dots < s_k = i$
 - dann ist $S[0, s_{k-1})$ splittable und $S[s_{k-1}, i)$ ein Wort



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i)$)

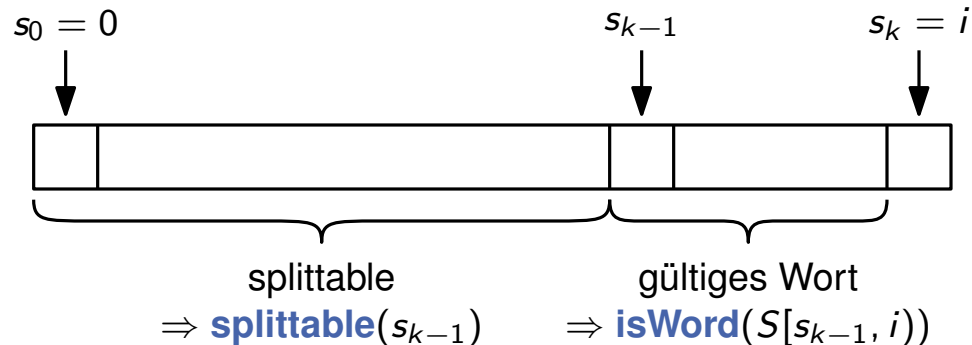
Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splittable**(i) = true.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splittable**(0) = true
- Hinrichtung: $S[0, i)$ splittable \Rightarrow **splittable**(i) = true
 - betrachte Lösung $0 = s_0 < \dots < s_k = i$
 - dann ist $S[0, s_{k-1})$ splittable und $S[s_{k-1}, i)$ ein Wort
 - nach I.V. gilt dann **splittable**(s_{k-1}) = true (und es gilt **isWord**(s_{k-1}, i) = true)



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i)$)

Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn $\text{splittable}(i) = \text{true}$.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und $\text{splittable}(0) = \text{true}$
- Rückrichtung: $S[0, i)$ splittable $\Leftarrow \text{splittable}(i) = \text{true}$

Erinnerung: Rekursive Definition von **splittable**

- $\text{splittable}(0) = \text{true}$
- $\text{splittable}(i) = \exists j \in [0, i) \text{ splittable}(j) \wedge \text{isWord}(S[j, i))$

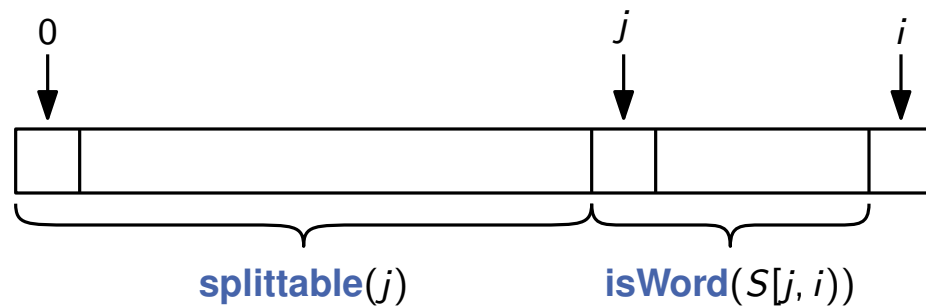
Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splittable**(i) = true.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splittable**(0) = true
- Rückrichtung: $S[0, i)$ splittable \Leftrightarrow **splittable**(i) = true
 - es gibt j , sodass **splittable**(j) und **isWord**($S[j, i)$)



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i)$)

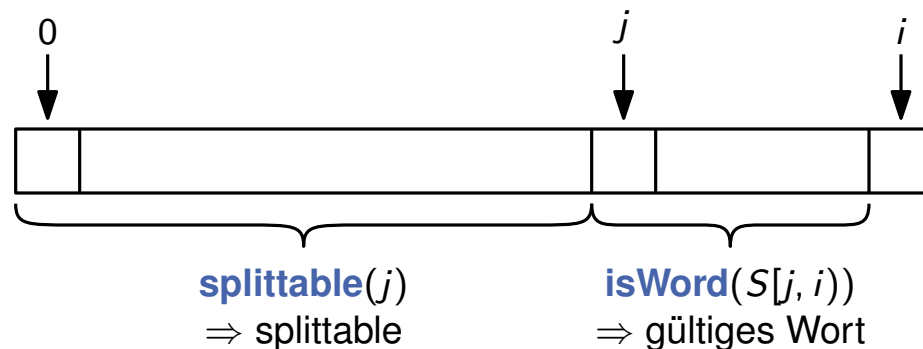
Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splittable**(i) = true.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splittable**(0) = true
- Rückrichtung: $S[0, i)$ splittable \Leftarrow **splittable**(i) = true
 - es gibt j , sodass **splittable**(j) und **isWord**($S[j, i)$)
 - nach I.V.: $S[0, j)$ ist splittable (außerdem ist $S[j, i)$ ein Wort)



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i)$)

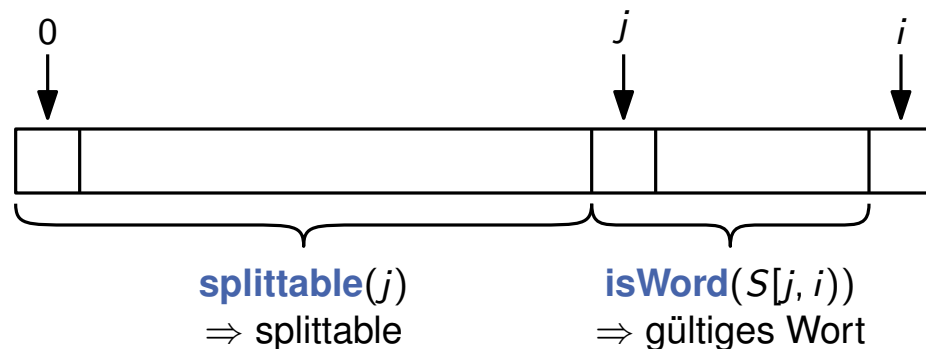
Korrektheit

Theorem

Der Teilstring $S[0, i)$ ist genau dann splittable, wenn **splittable**(i) = true.

Beweis: Induktion über i

- Induktionsanfang: der leere String ist splittable und **splittable**(0) = true
- Rückrichtung: $S[0, i)$ splittable \Leftarrow **splittable**(i) = true
 - es gibt j , sodass **splittable**(j) und **isWord**($S[j, i)$)
 - nach I.V.: $S[0, j)$ ist splittable (außerdem ist $S[j, i)$ ein Wort)
 - damit ist auch $S[0, i)$ splittable



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i)$)

Iterative Implementierung

isSplittable(*String S*)

```
s := Array of size  $n + 1$   
s[0] := true  
for  $i \in \{1, \dots, n\}$  do  
    s[i] = false  
    for  $j \in \{0, \dots, i - 1\}$  do  
        if s[j] and isWord(S[j, i]) then  
            s[i] := true  
return s[n]
```

Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size  $n + 1$ 
s[0] := true
for  $i \in \{1, \dots, n\}$  do
  s[i] = false
  for  $j \in \{0, \dots, i - 1\}$  do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

m	o	n	k	e	y	e	a	r	t	h	a	i	r
---	---	---	---	---	---	---	---	---	---	---	---	---	---

Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

m	o	n	k	e	y	e	a	r	t	h	a	i	r
---	---	---	---	---	---	---	---	---	---	---	---	---	---

s:

t													
---	--	--	--	--	--	--	--	--	--	--	--	--	--

Erinnerung: Rekursive Definition von splittable

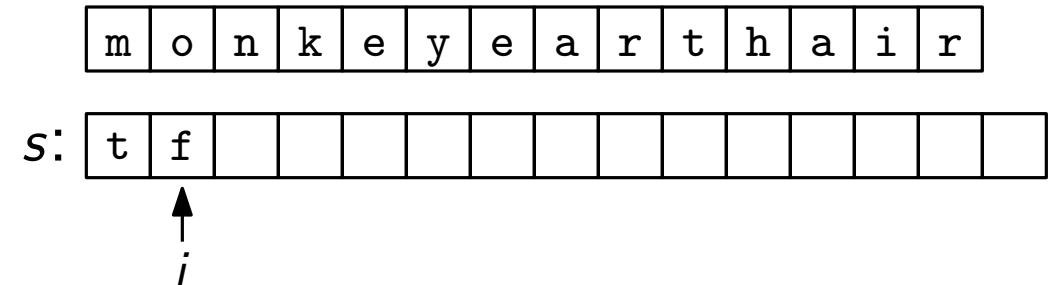
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**(S[j, i])

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**(S[j, i])

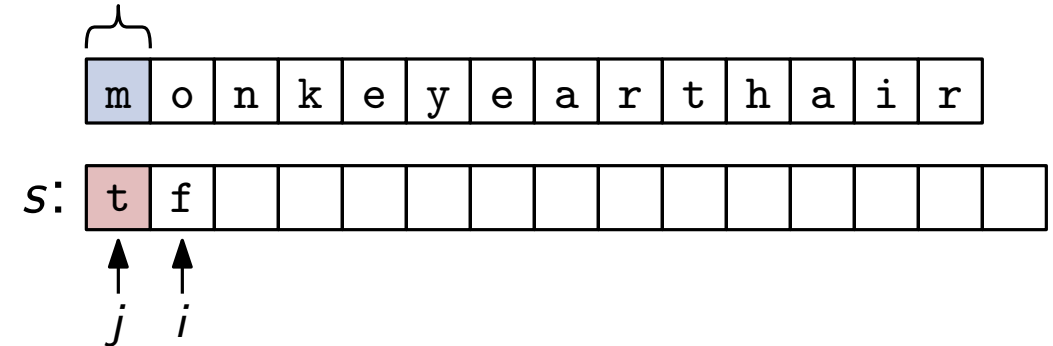
Iterative Implementierung

isSplittable(String S)

```

 $s :=$  Array of size  $n + 1$ 
 $s[0] :=$  true
for  $i \in \{1, \dots, n\}$  do
   $s[i] =$  false
  for  $j \in \{0, \dots, i - 1\}$  do
    if  $s[j]$  and isWord( $S[j, i]$ ) then
       $s[i] :=$  true
return  $s[n]$ 
  
```

isWord?



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

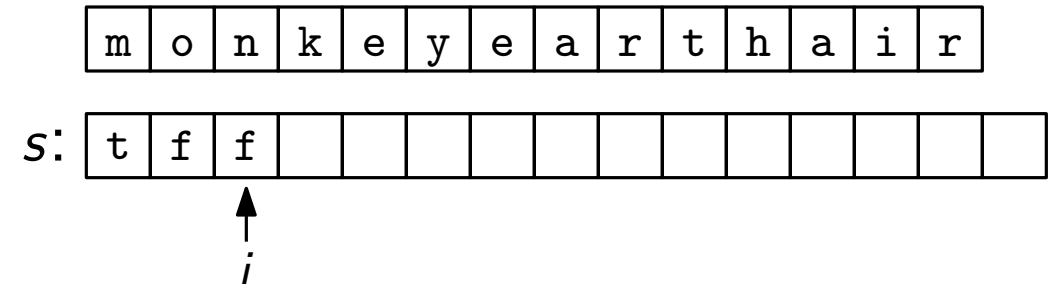
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

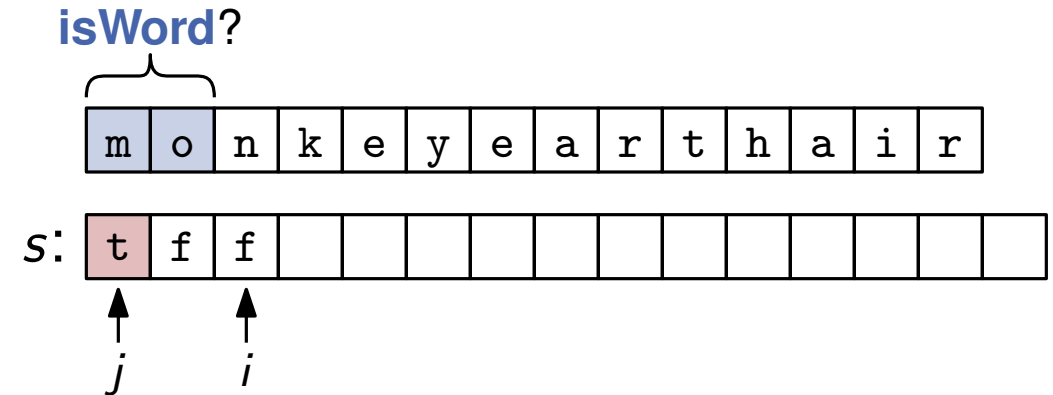
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(String S)

```

 $s :=$  Array of size  $n + 1$ 
 $s[0] :=$  true
for  $i \in \{1, \dots, n\}$  do
   $s[i] =$  false
  for  $j \in \{0, \dots, i - 1\}$  do
    if  $s[j]$  and isWord( $S[j, i]$ ) then
       $s[i] :=$  true
return  $s[n]$ 
  
```



Erinnerung: Rekursive Definition von splittable

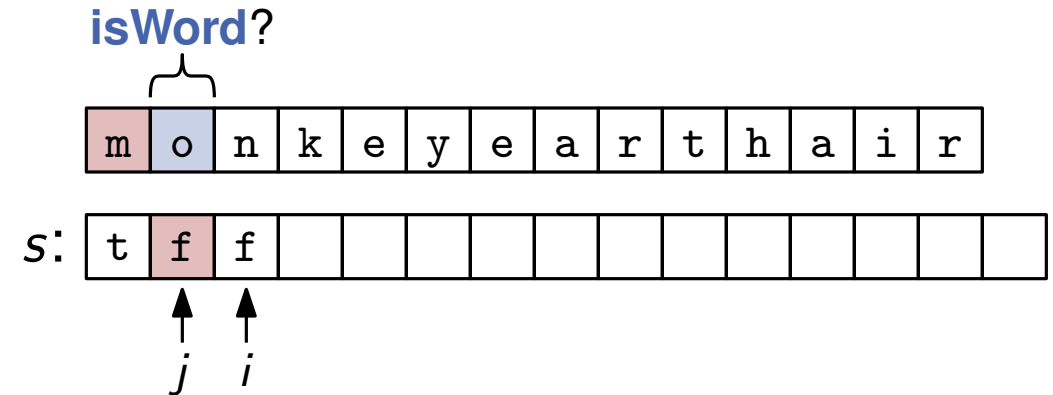
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

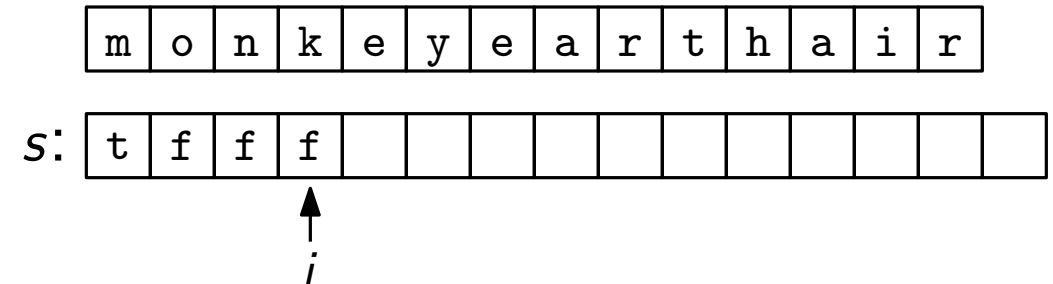
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i) \text{ splittable}(j) \wedge \text{isWord}(S[j, i])$

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von `splittable`

- `splittable(0) = true`
- `splittable(i) = ∃j ∈ [0, i) splittable(j) ∧ isWord(S[j, i])`

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

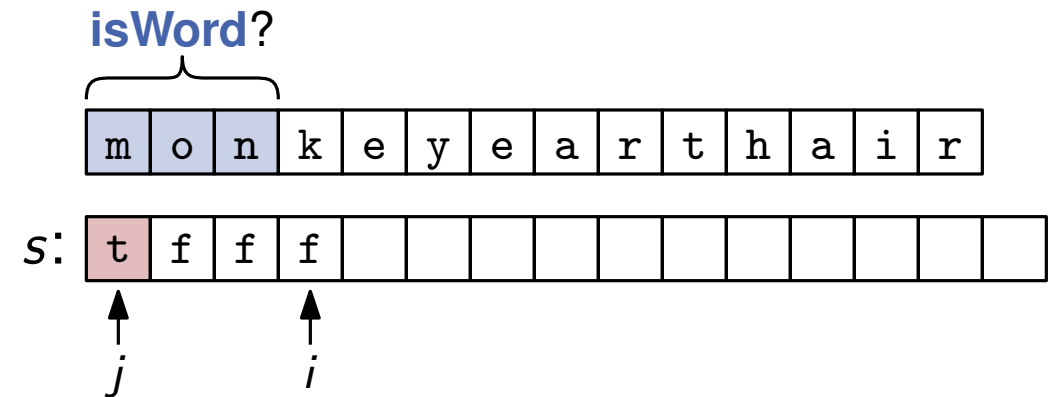
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

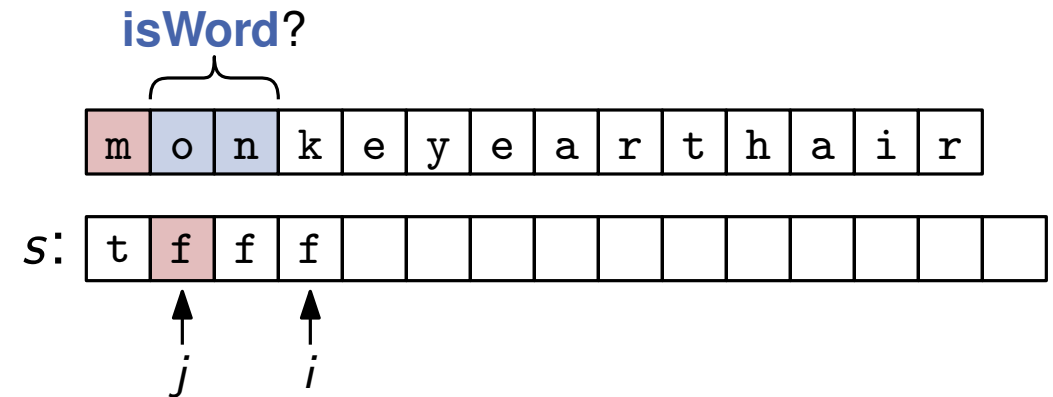
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

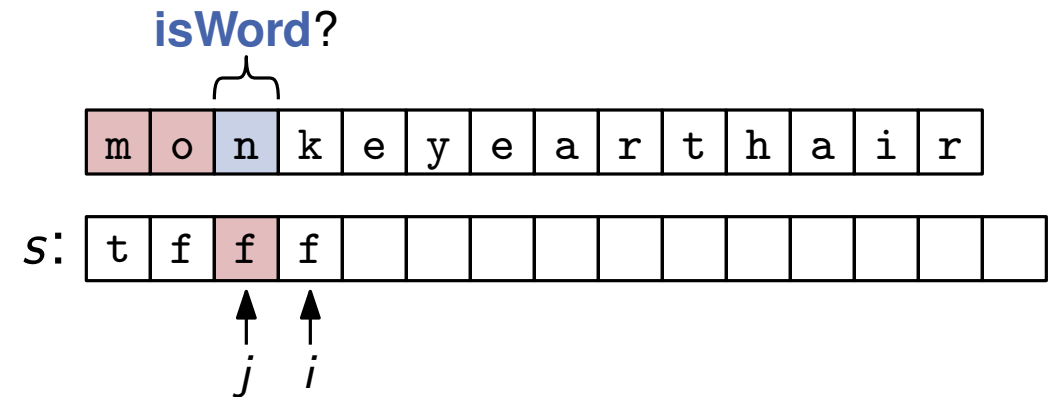
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

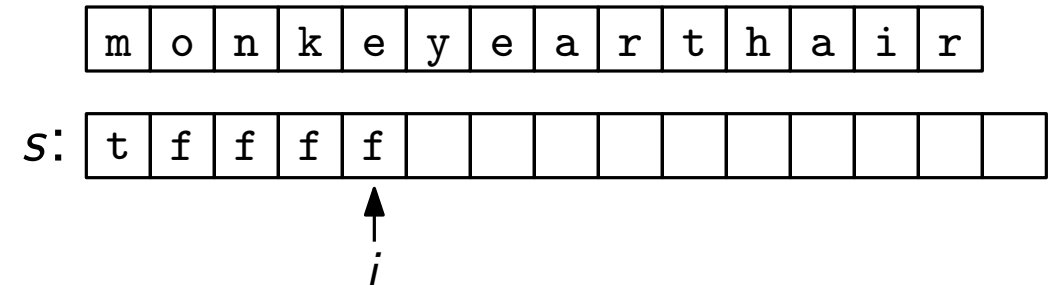
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**(S[j, i])

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size  $n + 1$ 
s[0] := true
for  $i \in \{1, \dots, n\}$  do
  s[i] = false
  for  $j \in \{0, \dots, i - 1\}$  do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(String S)

$s :=$ Array of size $n + 1$

$s[0] :=$ true

for $i \in \{1, \dots, n\}$ **do**

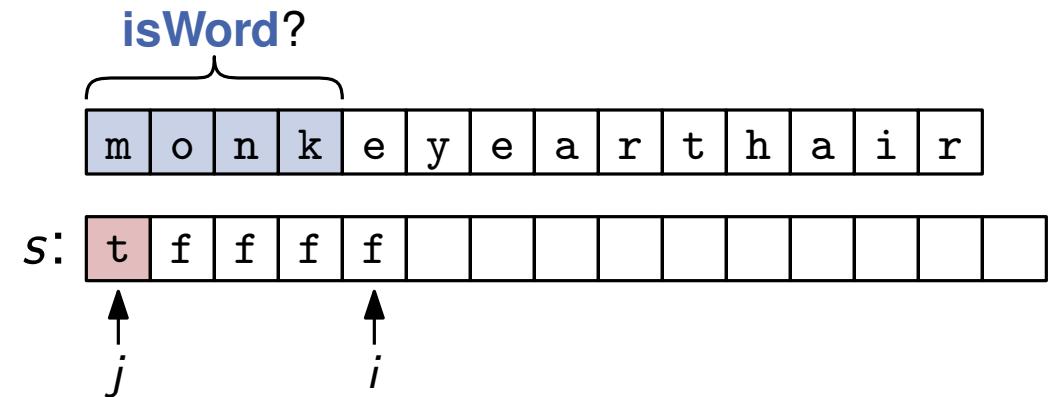
$s[i] =$ false

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] :=$ true

return $s[n]$



Erinnerung: Rekursive Definition von splittable

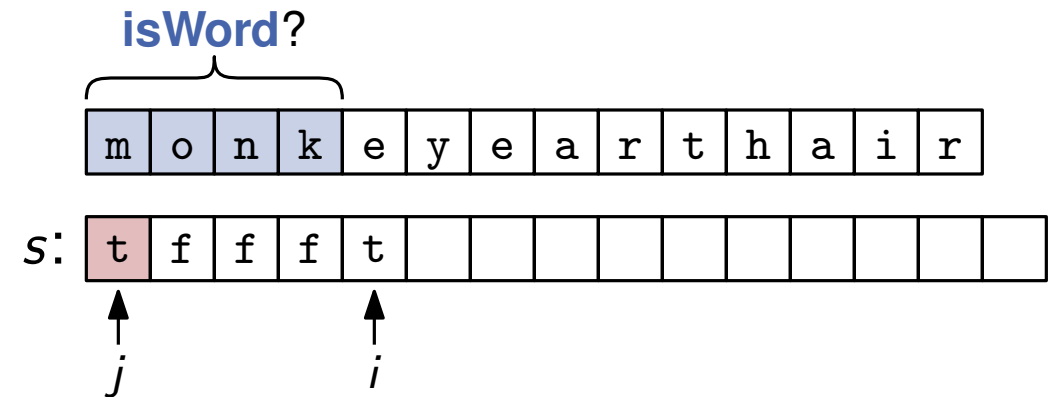
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

- `splittable(0) = true`
- `splittable(i) = ∃j ∈ [0, i) splittable(j) ∧ isWord(S[j, i])`

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

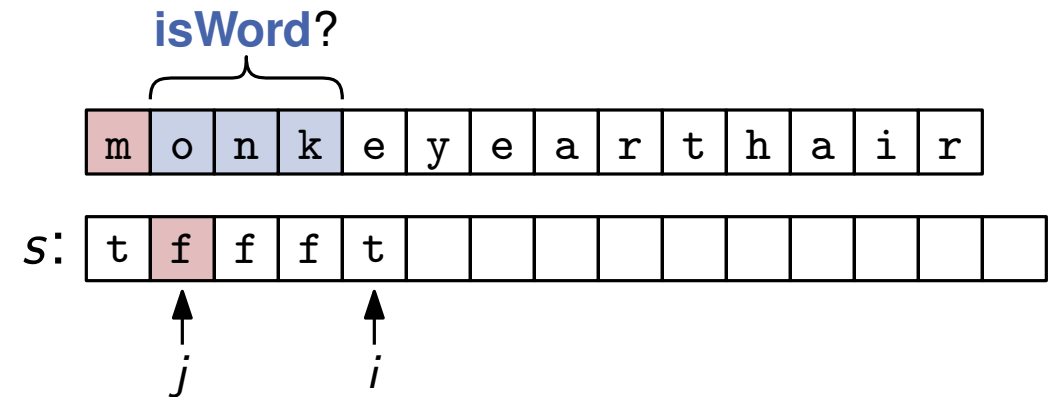
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(String S)

$s :=$ Array of size $n + 1$

$s[0] :=$ true

for $i \in \{1, \dots, n\}$ **do**

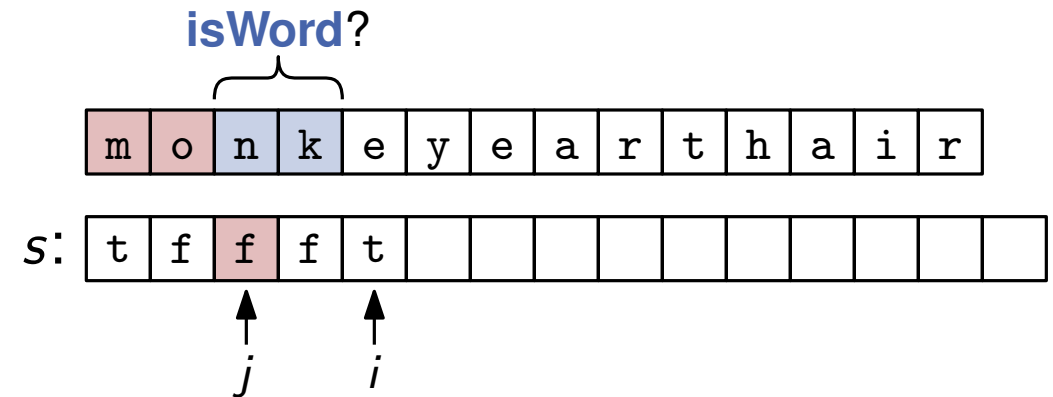
$s[i] =$ false

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] :=$ true

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(String S)

$s :=$ Array of size $n + 1$

$s[0] :=$ true

for $i \in \{1, \dots, n\}$ **do**

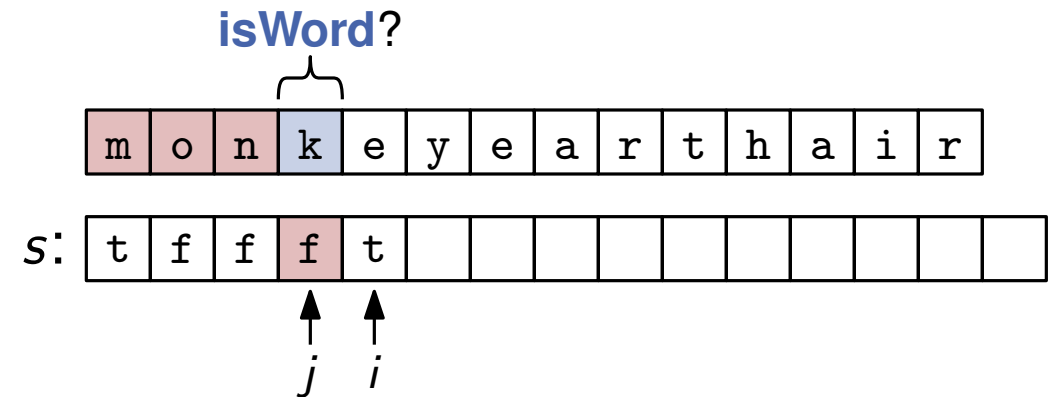
$s[i] =$ false

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] :=$ true

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

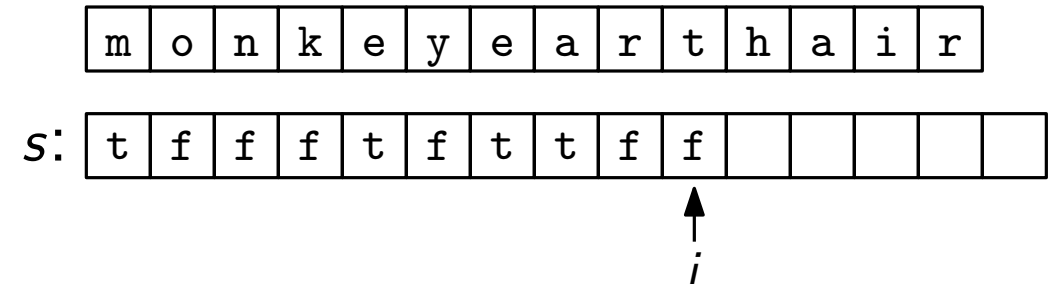
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

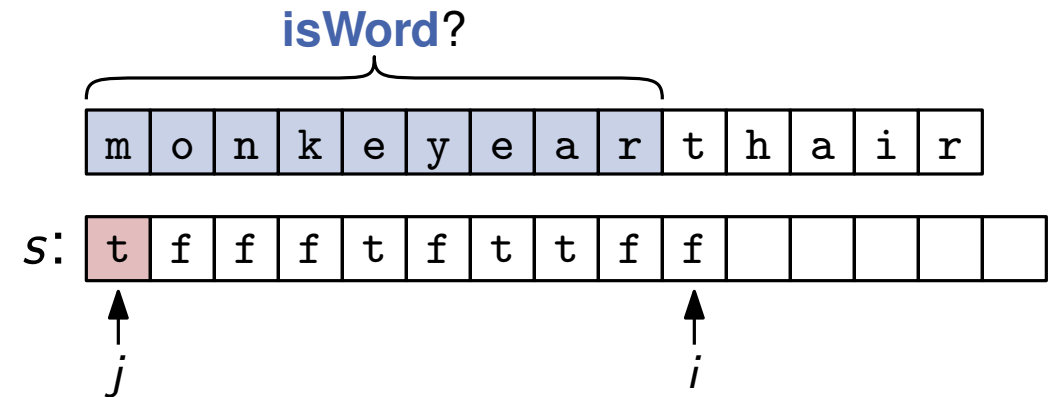
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

- `splittable(0) = true`
- `splittable(i) = ∃j ∈ [0, i) splittable(j) ∧ isWord(S[j, i])`

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

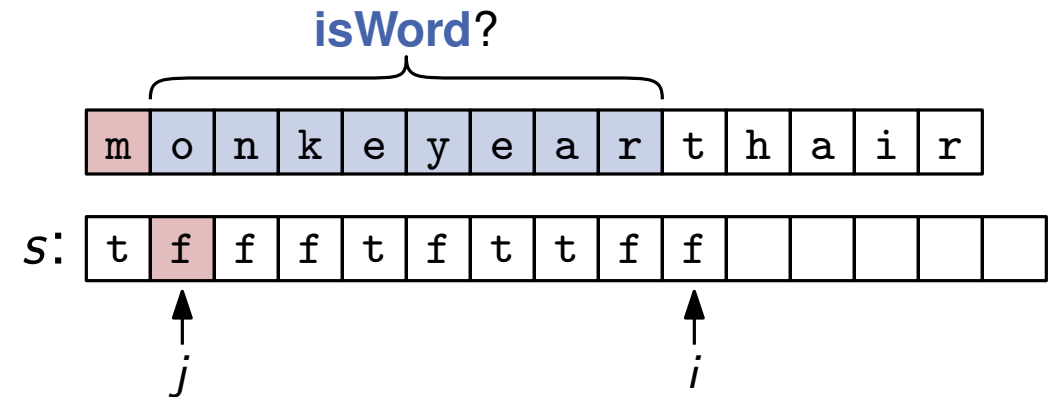
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

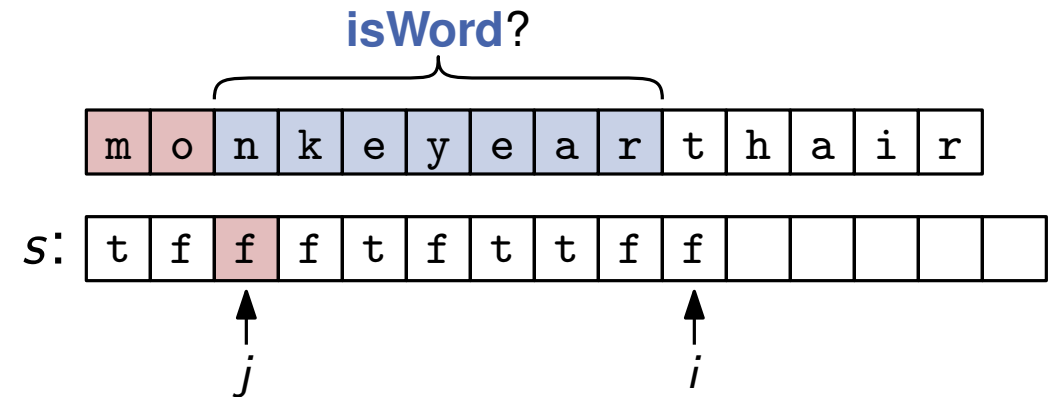
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

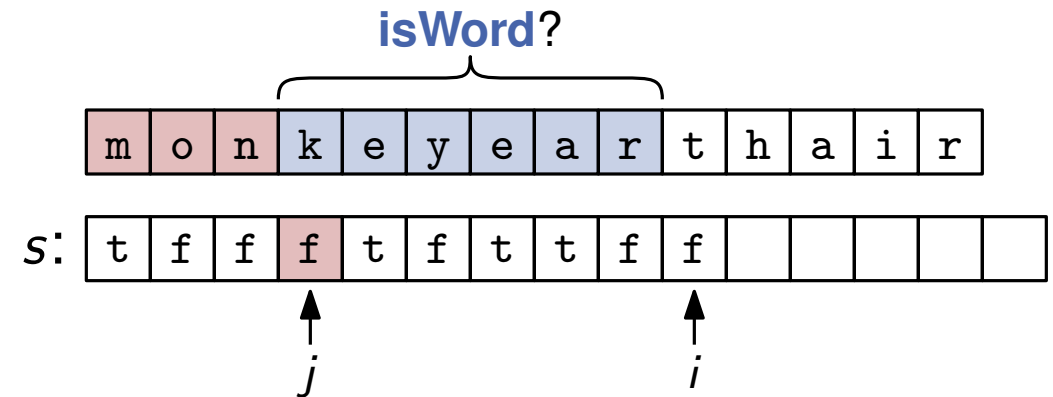
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(String S)

```

 $s :=$  Array of size  $n + 1$ 
 $s[0] :=$  true
for  $i \in \{1, \dots, n\}$  do
   $s[i] =$  false
  for  $j \in \{0, \dots, i - 1\}$  do
    if  $s[j]$  and isWord( $S[j, i]$ ) then
       $s[i] :=$  true
return  $s[n]$ 
  
```



Erinnerung: Rekursive Definition von `splittable`

- `splittable(0) = true`
- `splittable(i) = $\exists j \in [0, i)$ splittable(j) \wedge isWord(S[j, i])`

Iterative Implementierung

isSplittable(*String S*)

$s :=$ Array of size $n + 1$

$s[0] :=$ true

for $i \in \{1, \dots, n\}$ **do**

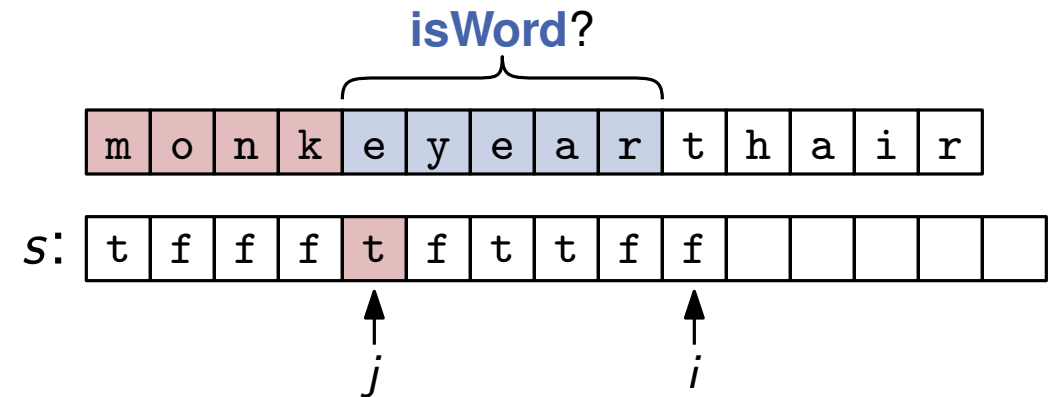
$s[i] =$ false

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] :=$ true

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

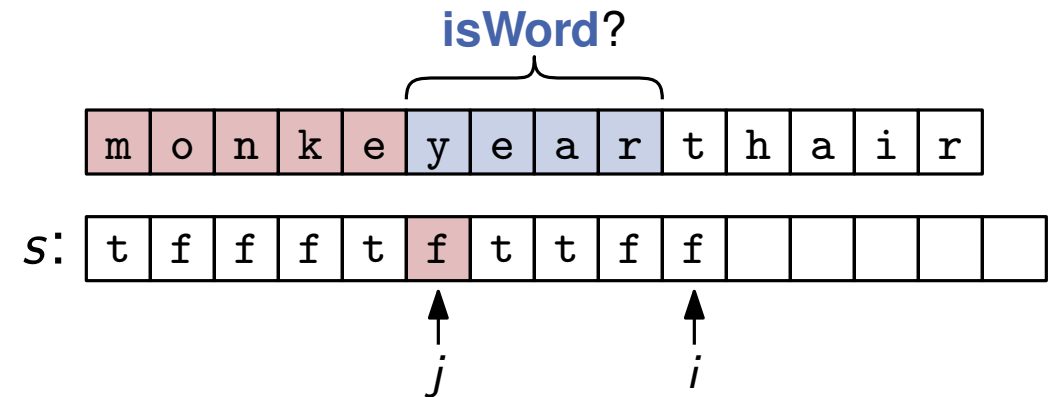
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

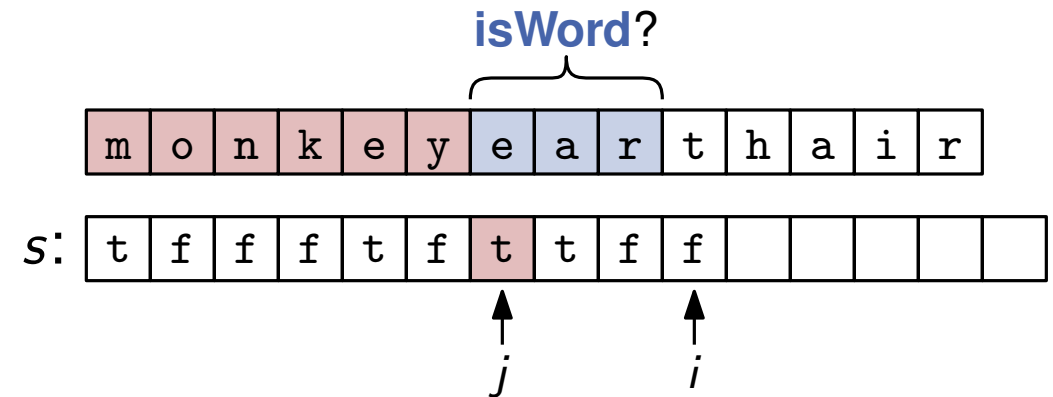
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

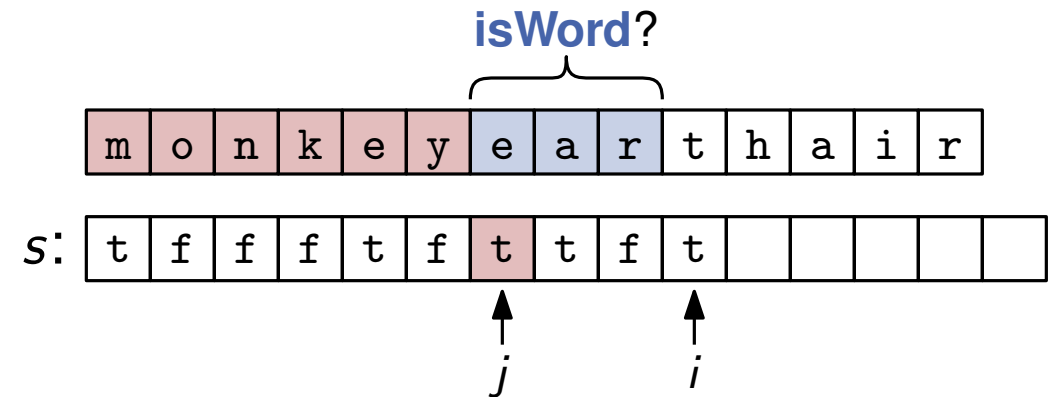
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

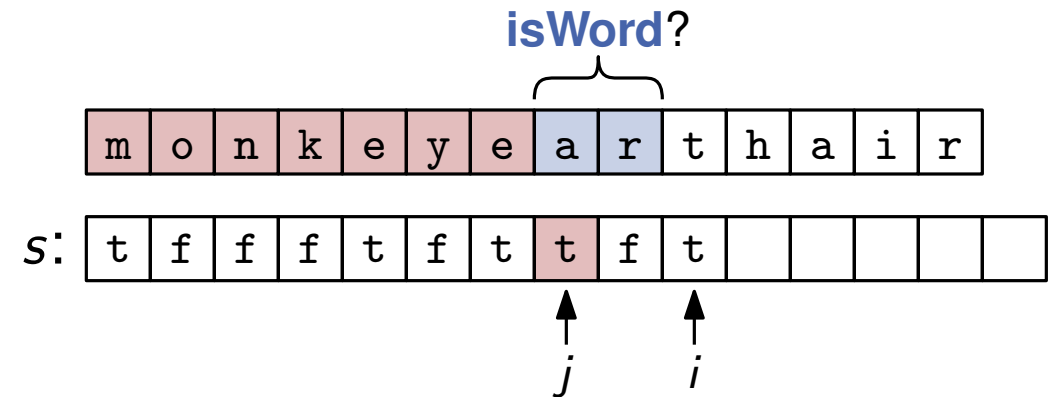
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

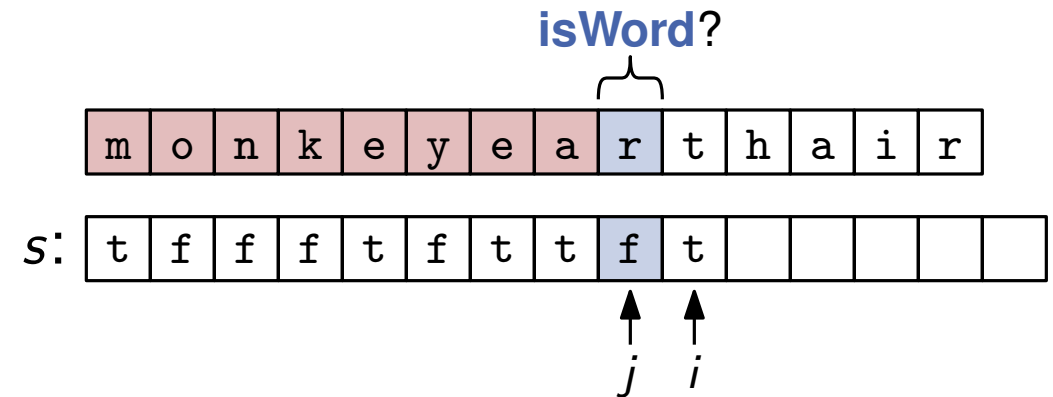
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

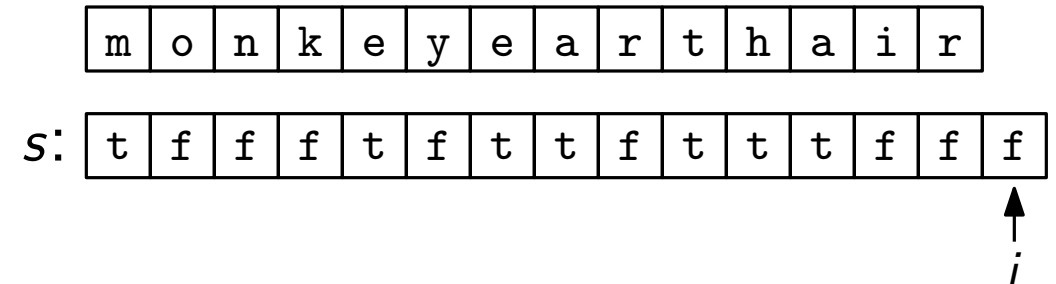
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

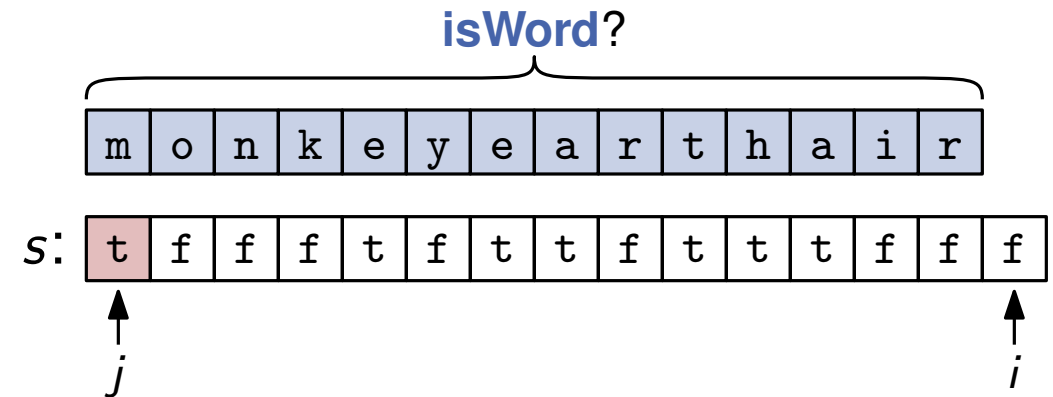
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

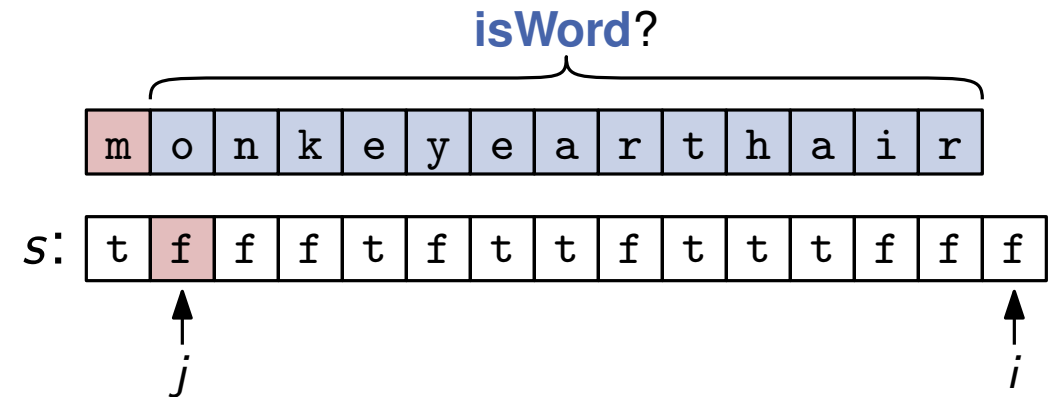
- `splittable(0) = true`
- `splittable(i) = ∃j ∈ [0, i) splittable(j) ∧ isWord(S[j, i])`

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

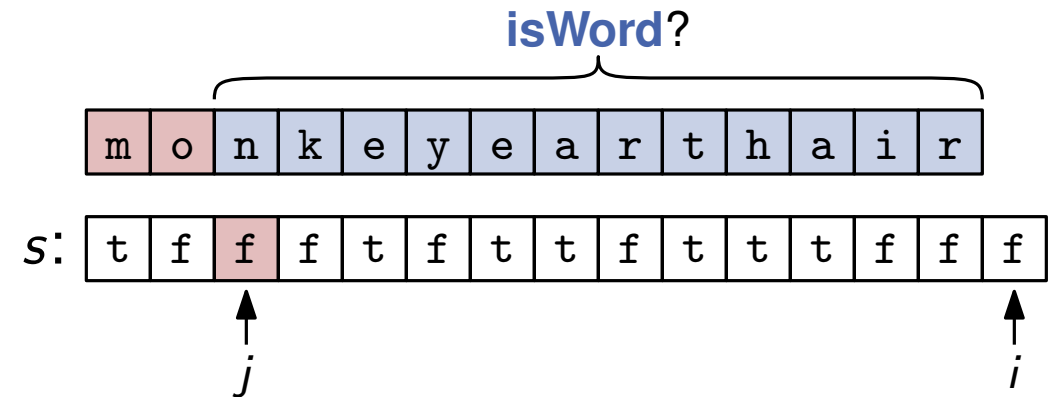
- `splittable(0) = true`
- `splittable(i) = ∃j ∈ [0, i) splittable(j) ∧ isWord(S[j, i])`

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**(S[j, i])

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

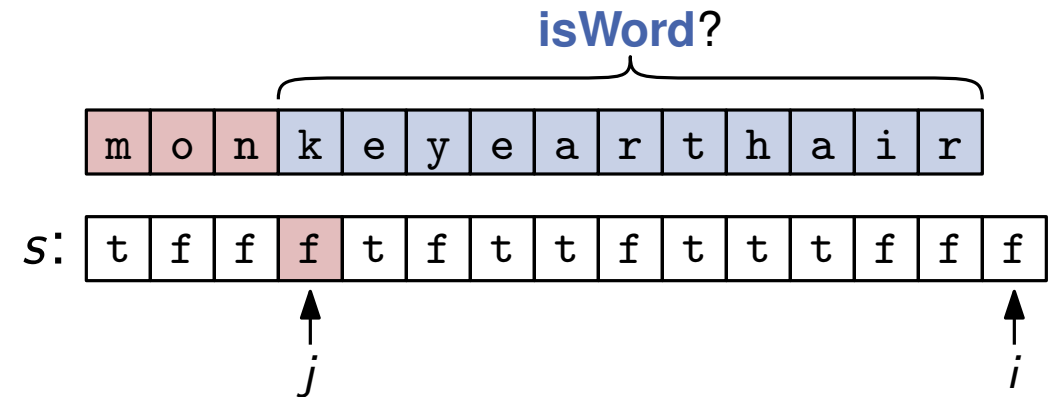
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

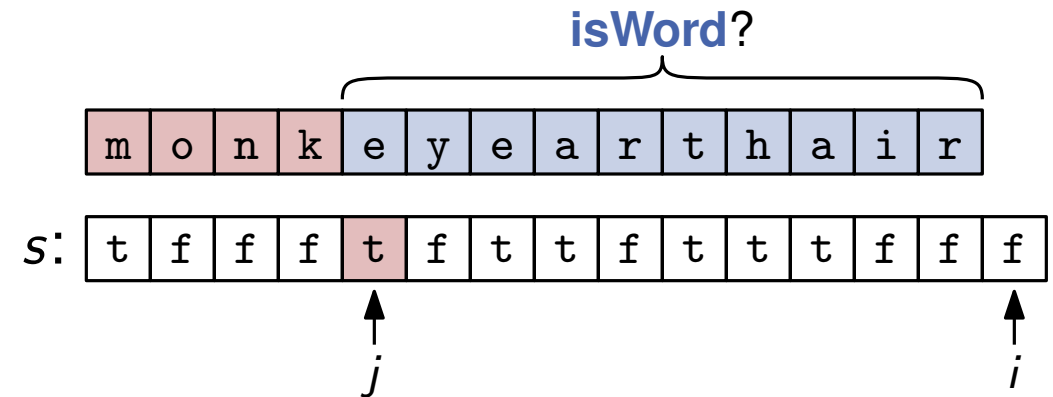
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

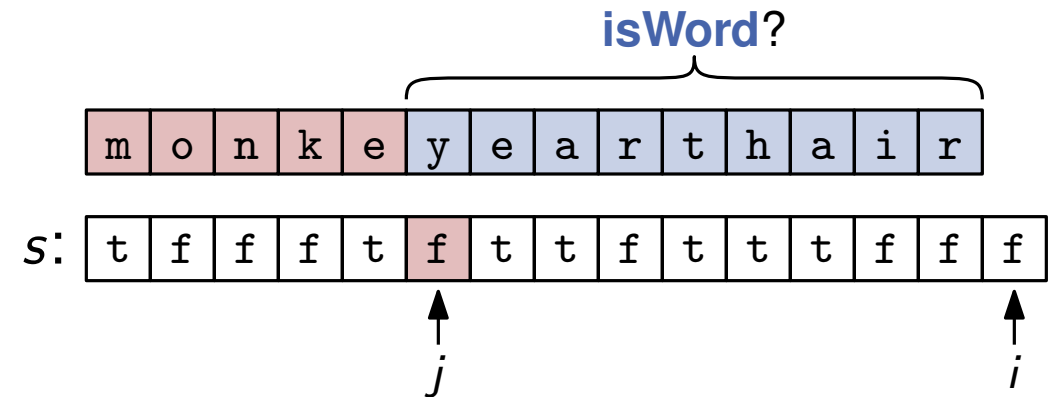
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

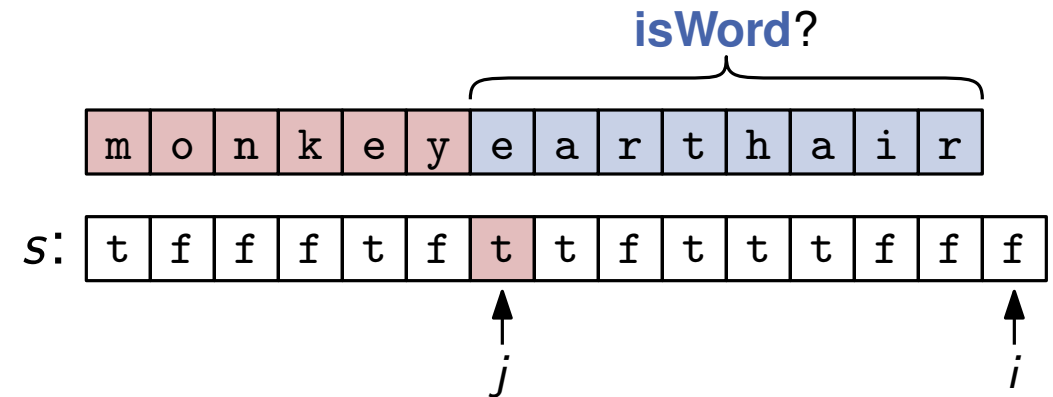
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

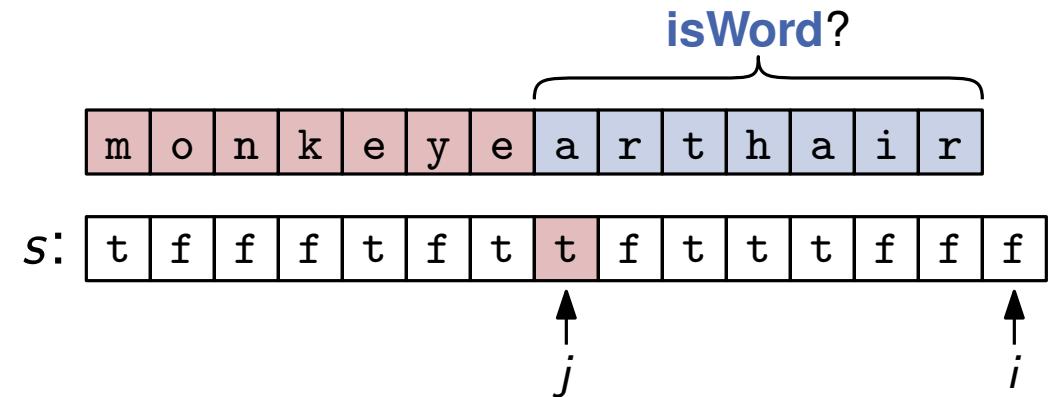
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von `splittable`

- `splittable(0) = true`
- `splittable(i) = $\exists j \in [0, i)$ splittable(j) \wedge isWord(S[j, i])`

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

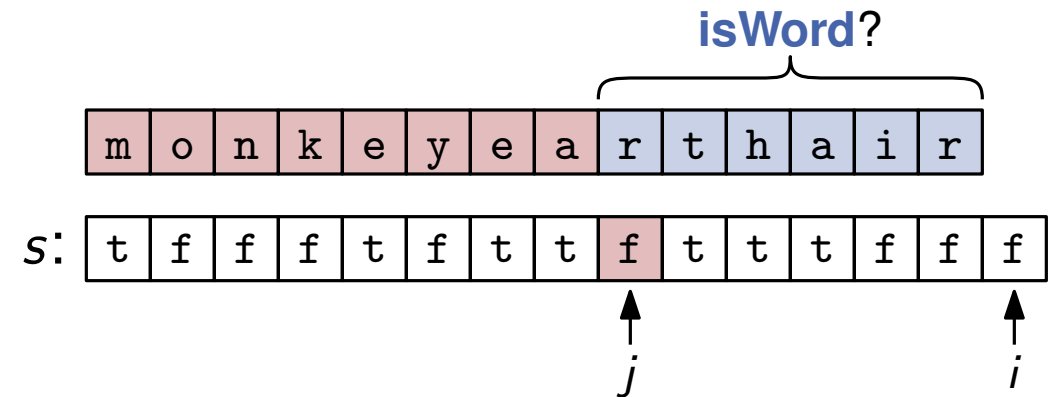
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von `splittable`

- `splittable(0) = true`
- `splittable(i) = $\exists j \in [0, i)$ splittable(j) \wedge isWord(S[j, i])`

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

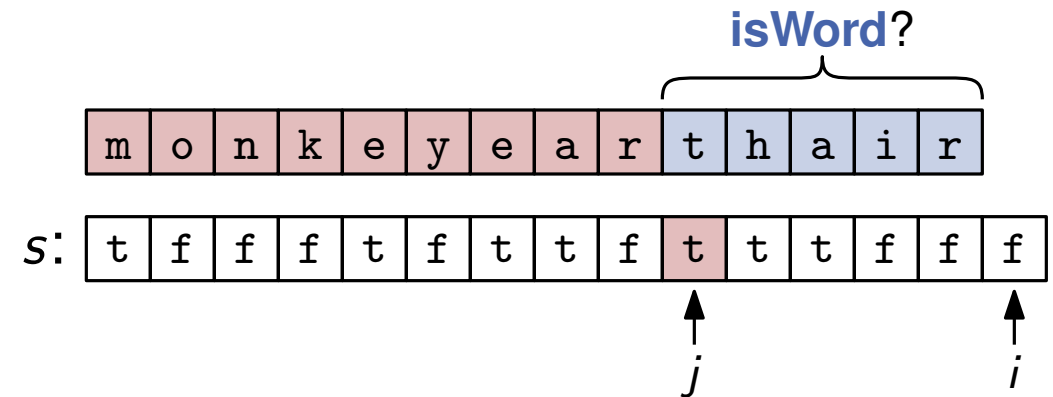
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

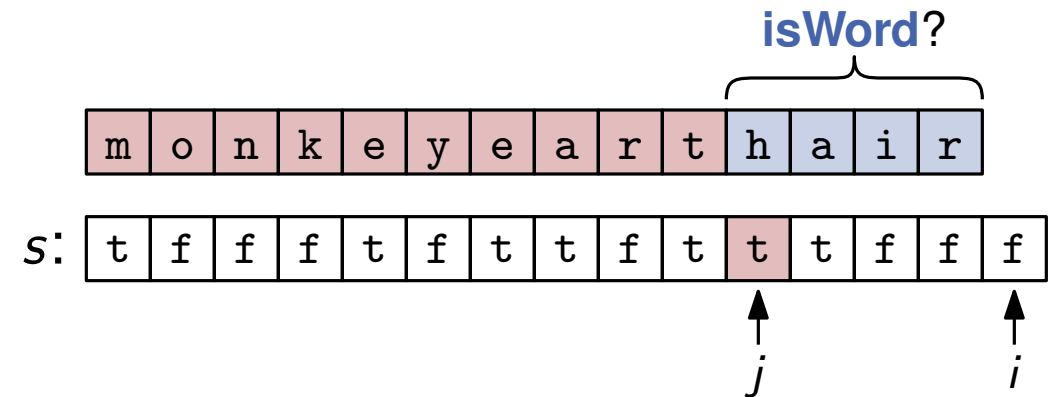
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

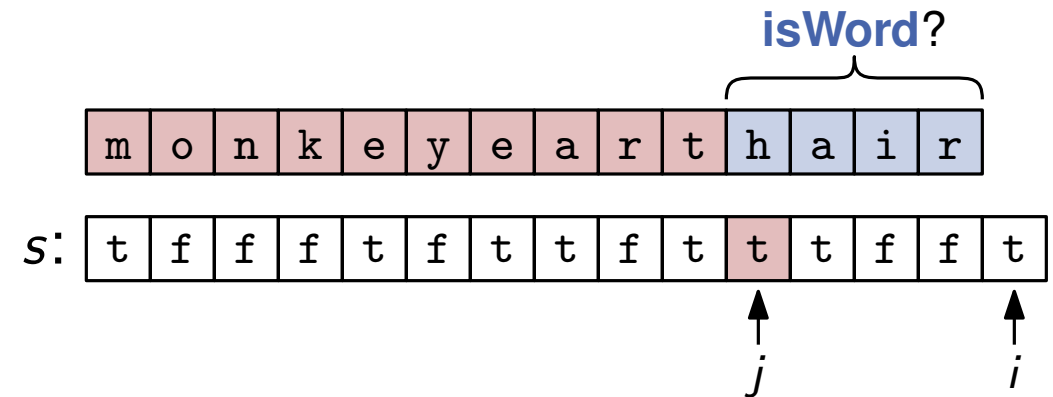
- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(String S)

$s :=$ Array of size $n + 1$

$s[0] :=$ true

for $i \in \{1, \dots, n\}$ **do**

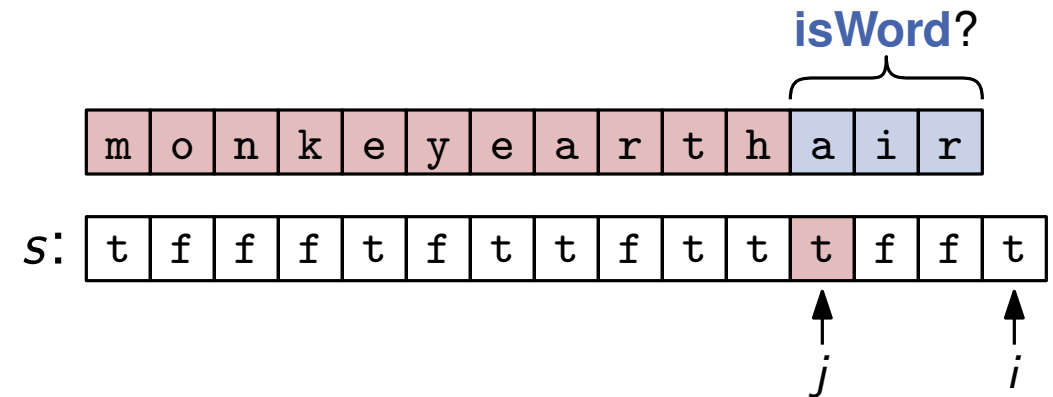
$s[i] =$ false

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] :=$ true

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- `splittable(0) = true`
- `splittable(i) = $\exists j \in [0, i)$ splittable(j) \wedge isWord(S[j, i])`

Iterative Implementierung

isSplittable(String S)

$s :=$ Array of size $n + 1$

$s[0] :=$ true

for $i \in \{1, \dots, n\}$ **do**

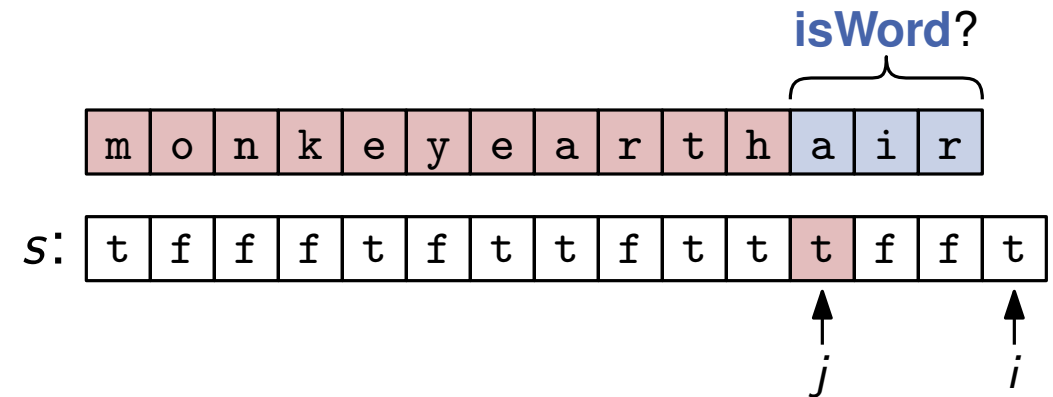
$s[i] =$ false

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] :=$ true

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

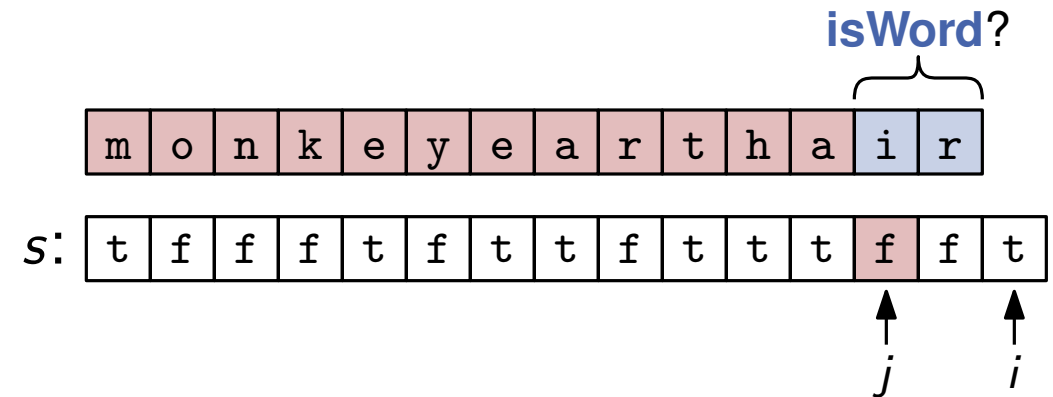
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von `splittable`

- `splittable`(0) = true
- `splittable`(i) = $\exists j \in [0, i)$ `splittable`(j) \wedge `isWord`($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

$s := \text{Array of size } n + 1$

$s[0] := \text{true}$

for $i \in \{1, \dots, n\}$ **do**

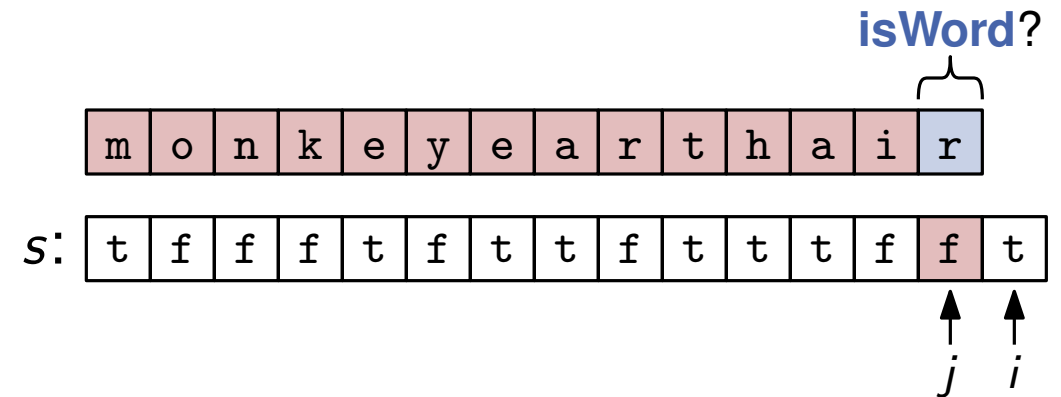
$s[i] = \text{false}$

for $j \in \{0, \dots, i - 1\}$ **do**

if $s[j]$ **and** **isWord**($S[j, i]$) **then**

$s[i] := \text{true}$

return $s[n]$



Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**($S[j, i]$)

Iterative Implementierung

isSplittable(*String S*)

```

s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

m	o	n	k	e	y	e	a	r	t	h	a	i	r
---	---	---	---	---	---	---	---	---	---	---	---	---	---

s:

t	f	f	f	t	f	t	t	f	t	t	t	f	f	t
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(*String S*)

```
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Erinnerung: Rekursive Definition von splittable

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**(S[j, i])

Iterative Implementierung

isSplittable(*String S*)

```

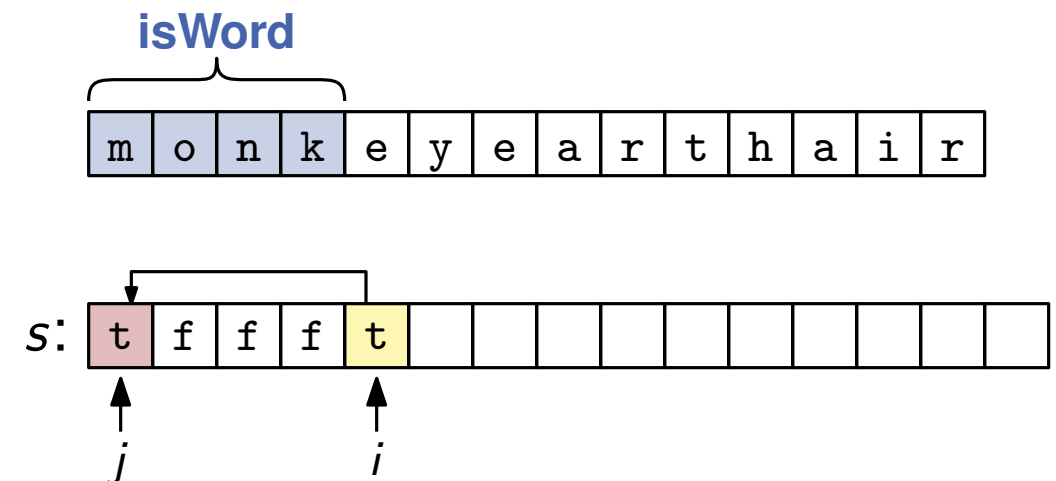
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes *i* wegen welchem *j* wir $s[i] := \text{true}$ gesetzt haben



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(*String S*)

```

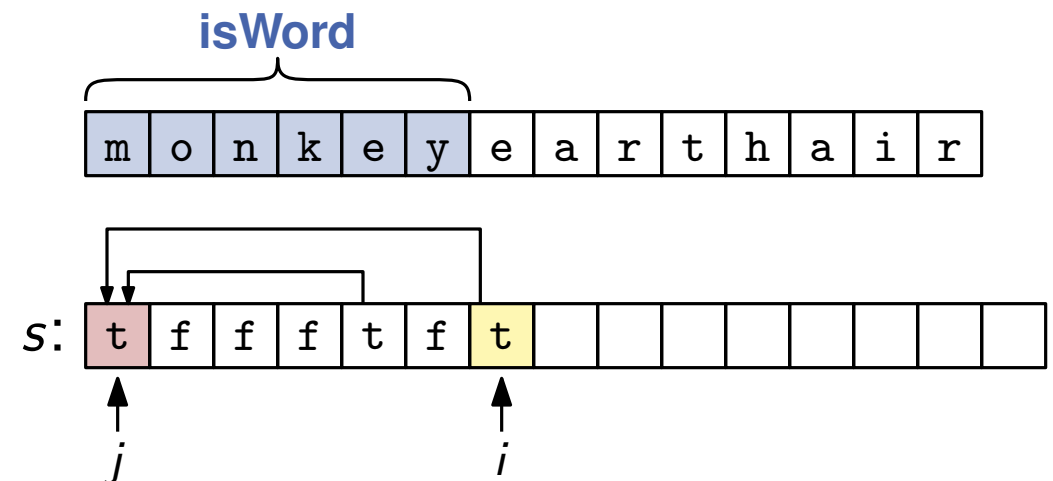
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes *i* wegen welchem *j* wir $s[i] := \text{true}$ gesetzt haben



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i) \text{ splittable}(j) \wedge \text{isWord}(S[j, i])$

Iterative Implementierung

isSplittable(*String S*)

```

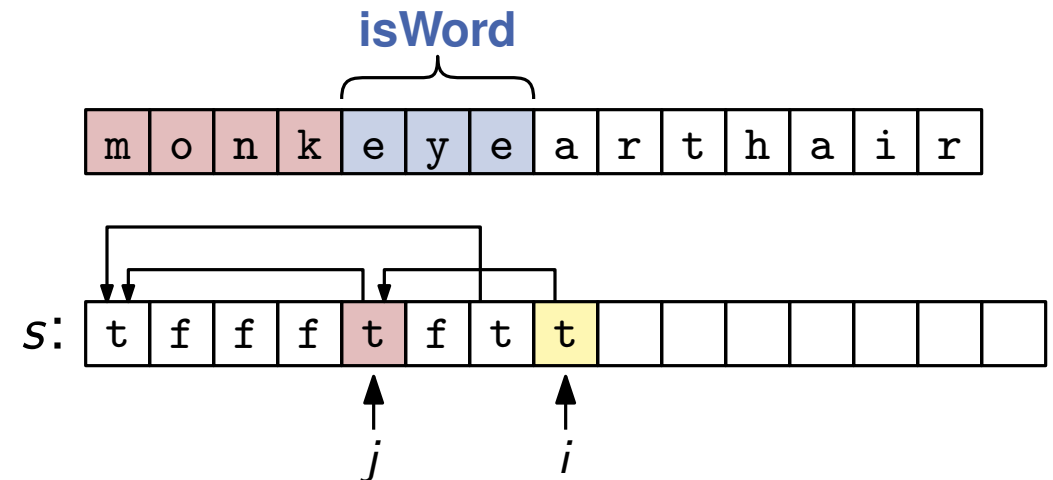
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes *i* wegen welchem *j* wir $s[i] := \text{true}$ gesetzt haben



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(*String S*)

```

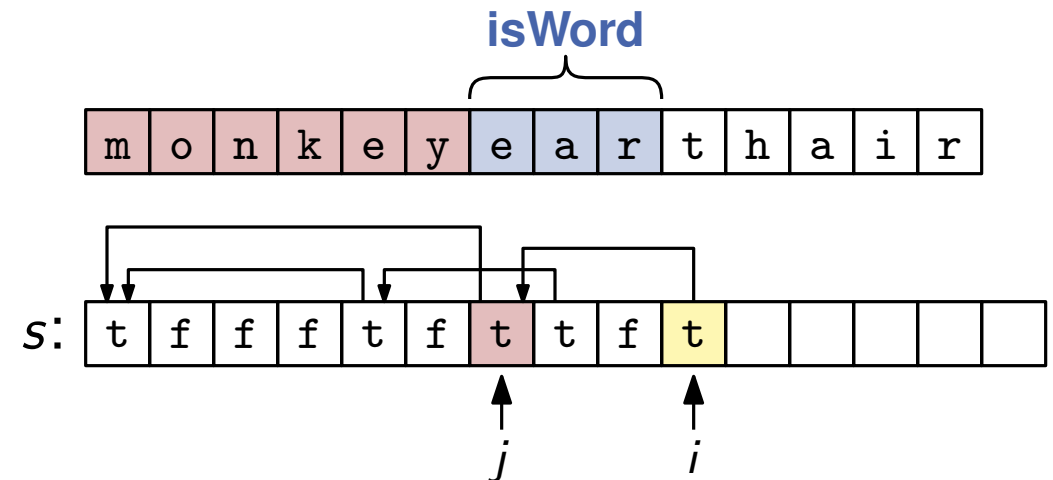
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes *i* wegen welchem *j* wir $s[i] := \text{true}$ gesetzt haben



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**(S[j, i])

Iterative Implementierung

isSplittable(*String S*)

```

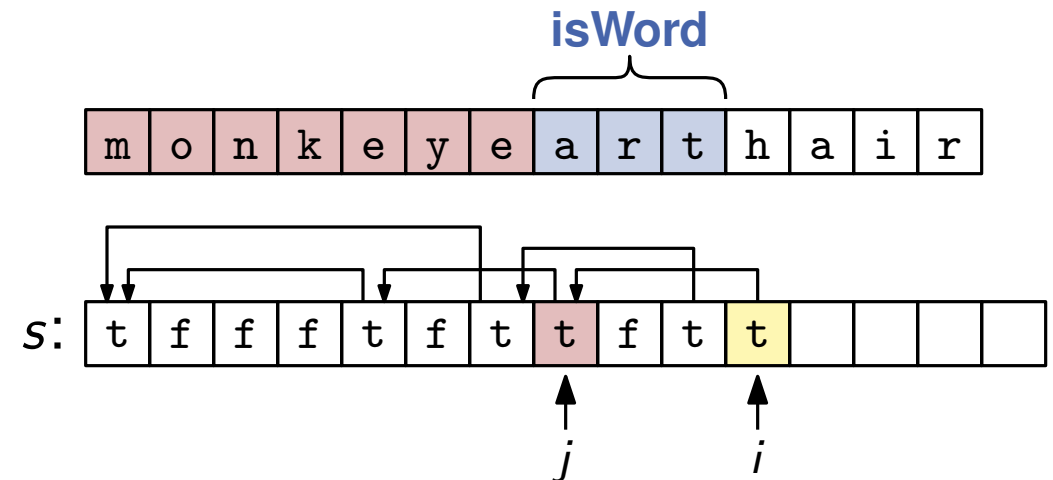
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes *i* wegen welchem *j* wir $s[i] := \text{true}$ gesetzt haben



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(*String S*)

```

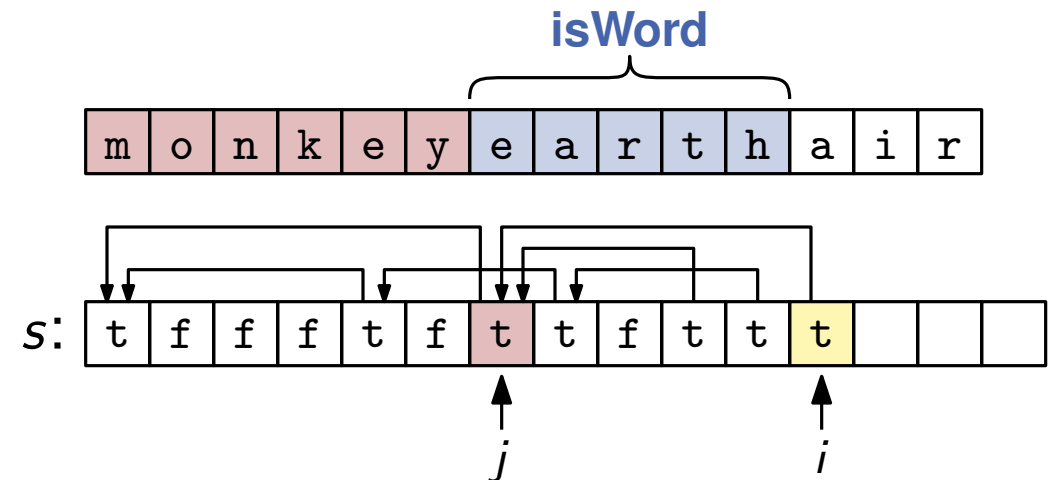
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes *i* wegen welchem *j* wir $s[i] := \text{true}$ gesetzt haben



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(i) = $\exists j \in [0, i)$ **splittable**(j) \wedge **isWord**(S[j, i])

Iterative Implementierung

isSplittable(*String S*)

```

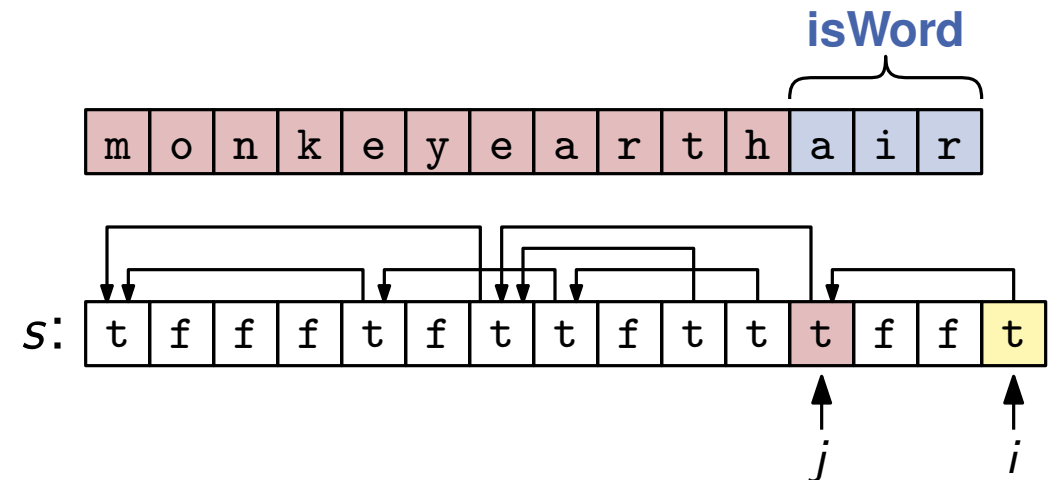
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
  return s[n]
  
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes *i* wegen welchem *j* wir *s*[*i*] := true gesetzt haben



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

Iterative Implementierung

isSplittable(*String S*)

```

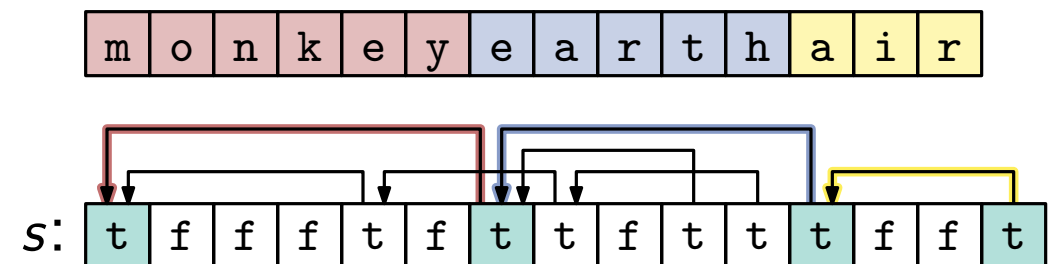
s := Array of size n + 1
s[0] := true
for i ∈ {1, ..., n} do
  s[i] = false
  for j ∈ {0, ..., i - 1} do
    if s[j] and isWord(S[j, i]) then
      s[i] := true
return s[n]
  
```

Laufzeit

- $\Theta(n^2)$ Aufrufe von **isWord** (also: polynomiell)

Bekommen wir auch eine Lösung?

- speichere für jedes *i* wegen welchem *j* wir *s*[*i*] := true gesetzt haben
- verfolge diese Pointer rückwärts von *n* aus



Erinnerung: Rekursive Definition von **splittable**

- **splittable**(0) = true
- **splittable**(*i*) = $\exists j \in [0, i)$ **splittable**(*j*) \wedge **isWord**(*S*[*j*, *i*])

In drei Schritten zum dynamischen Programm

Schritt 1: Spezifikation der Teilprobleme

- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?
- Beispiel: Ist $S[0, i)$ splittable?

Schritt 2: Aufstellung der Rekurrenz für Teillösungen

- Wie berechnen wir neue aus alten Zwischenergebnissen?
- Korrektheit: Induktion
- Beispiel: Lösung für i dank Lösungen für $0, \dots, i - 1$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

In drei Schritten zum dynamischen Programm

Schritt 1: Spezifikation der Teilprobleme

- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?
- Beispiel: Ist $S[0, i)$ splittable?

schwer
erfordert Kreativität
ggf. richtige Idee nötig
interagiert mit Schritt 2

Schritt 2: Aufstellung der Rekurrenz für Teillösungen

- Wie berechnen wir neue aus alten Zwischenergebnissen?
- Korrektheit: Induktion
- Beispiel: Lösung für i dank Lösungen für $0, \dots, i - 1$

nicht superschwer

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

leicht
(mit etwas Übung)

Longest Common Subsequence

Problem: LCS

Gegeben Folgen $\langle a_1, \dots, a_n \rangle$ und $\langle b_1, \dots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?

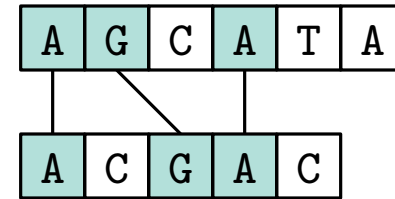
A	G	C	A	T	A
---	---	---	---	---	---

A	C	G	A	C
---	---	---	---	---

Longest Common Subsequence

Problem: LCS

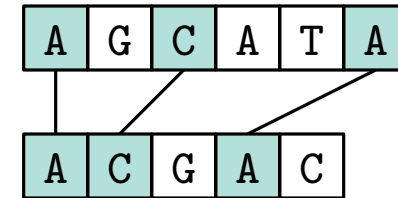
Gegeben Folgen $\langle a_1, \dots, a_n \rangle$ und $\langle b_1, \dots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?



Longest Common Subsequence

Problem: LCS

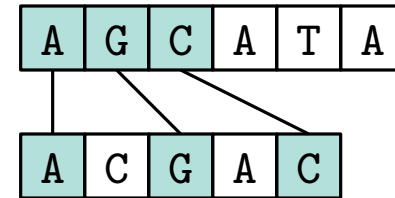
Gegeben Folgen $\langle a_1, \dots, a_n \rangle$ und $\langle b_1, \dots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?



Longest Common Subsequence

Problem: LCS

Gegeben Folgen $\langle a_1, \dots, a_n \rangle$ und $\langle b_1, \dots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?



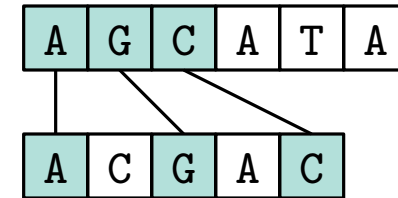
Longest Common Subsequence

Problem: LCS

Gegeben Folgen $\langle a_1, \dots, a_n \rangle$ und $\langle b_1, \dots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?

Schritt 1: Spezifikation der Teilprobleme

- Wie lang ist die LCS von $\langle a_1, \dots, a_i \rangle$ und $\langle b_1, \dots, b_j \rangle$?



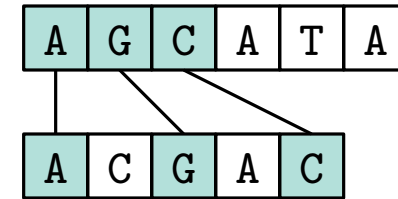
Longest Common Subsequence

Problem: LCS

Gegeben Folgen $\langle a_1, \dots, a_n \rangle$ und $\langle b_1, \dots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?

Schritt 1: Spezifikation der Teilprobleme

- Wie lang ist die LCS von $\langle a_1, \dots, a_i \rangle$ und $\langle b_1, \dots, b_j \rangle$?
- $i = n$ und $j = m$ liefert Gesamtlösung
- vereinfachte Sichtweise: nur Größe der Lösung (Berechnung der Lösung selbst heben wir uns für Schritt 3 auf)



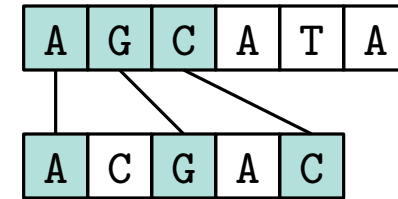
Longest Common Subsequence

Problem: LCS

Gegeben Folgen $\langle a_1, \dots, a_n \rangle$ und $\langle b_1, \dots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?

Schritt 1: Spezifikation der Teilprobleme

- Wie lang ist die LCS von $\langle a_1, \dots, a_i \rangle$ und $\langle b_1, \dots, b_j \rangle$?
- $i = n$ und $j = m$ liefert Gesamtlösung
- vereinfachte Sichtweise: nur Größe der Lösung (Berechnung der Lösung selbst heben wir uns für Schritt 3 auf)



Schritt 2: Aufstellung der Rekurrenz

- Ziel: $\mathbf{LCS}(i, j)$ bezeichnet Länge der LCS von $\langle a_1, \dots, a_i \rangle$ und $\langle b_1, \dots, b_j \rangle$
- $\mathbf{LCS}(i, 0) = \mathbf{LCS}(0, j) = 0$

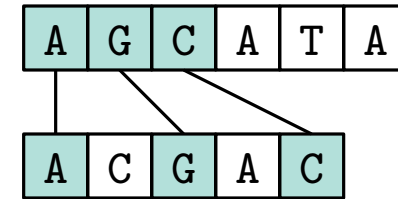
Longest Common Subsequence

Problem: LCS

Gegeben Folgen $\langle a_1, \dots, a_n \rangle$ und $\langle b_1, \dots, b_m \rangle$, was ist die längste gemeinsame Teilfolge?

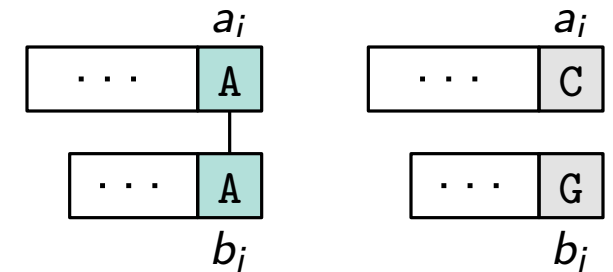
Schritt 1: Spezifikation der Teilprobleme

- Wie lang ist die LCS von $\langle a_1, \dots, a_i \rangle$ und $\langle b_1, \dots, b_j \rangle$?
- $i = n$ und $j = m$ liefert Gesamtlösung
- vereinfachte Sichtweise: nur Größe der Lösung (Berechnung der Lösung selbst heben wir uns für Schritt 3 auf)



Schritt 2: Aufstellung der Rekurrenz

- Ziel: $\mathbf{LCS}(i, j)$ bezeichnet Länge der LCS von $\langle a_1, \dots, a_i \rangle$ und $\langle b_1, \dots, b_j \rangle$
- $\mathbf{LCS}(i, 0) = \mathbf{LCS}(0, j) = 0$
- $\mathbf{LCS}(i, j) = \begin{cases} \mathbf{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\mathbf{LCS}(i, j-1), \mathbf{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$
- Korrektheit: folgt induktiv



Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$

- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

	$j \rightarrow$	0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0							
	1	A						
	2	C						
	3	G						
	4	A						
	5	C						

Longest Common Subsequence: Schritt 3

Rekurrenz

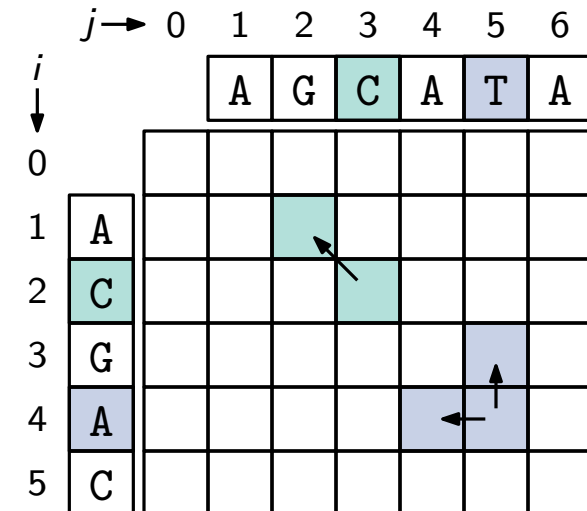
- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?



Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0							
1	A							
2	C							
3	G							
4	A							
5	C							

The table shows a grid for calculating the Longest Common Subsequence (LCS) between the strings "AGCAT" (top row) and "ACG" (left column). The grid cells are colored based on the value of the LCS at that position:

- Light blue cells: (2,2), (3,3), (4,4)
- Light green cells: (1,3), (2,4)
- Light purple cells: (3,5), (4,5)

 Arrows indicate the path of the LCS: from (4,5) to (3,5), then to (3,4), then to (2,4), then to (1,3), and finally to (1,2).

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)

	$j \rightarrow$	0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0						
2	C	0						
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

$j \rightarrow$	0	1	2	3	4	5	6
$i \downarrow$		A	G	C	A	T	A
0	0	0	0	0	0	0	0
1	A	0	1				
2	C	0					
3	G	0					
4	A	0					
5	C	0					

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

	$j \rightarrow$	0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	-1				
2	C	0						
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
	1	A	0	1	←1	←1		
	2	C	0					
	3	G	0					
	4	A	0					
	5	C	0					

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1		
2	C	0						
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	
2	C	0						
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0						
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
	1	A	0	1	←1	←1	1	←1
	2	C	0	1				
	3	G	0					
	4	A	0					
	5	C	0					

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1				
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2			
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$												
		0	1	2	3	4	5	6						
		<table border="1"> <tr> <td>A</td> <td>G</td> <td>C</td> <td>A</td> <td>T</td> <td>A</td> </tr> </table>							A	G	C	A	T	A
A	G	C	A	T	A									
$i \downarrow$	0	0	0	0	0	0	0	0						
1	A	0	1	1	1	1	1	1						
2	C	0	1	1	2	2								
3	G	0												
4	A	0												
5	C	0												

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	←1	←1	←1	1	←1
2	C	0	1	←1	2	←2	←2	←2
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0						
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1					
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2				
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		j \rightarrow						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
i \downarrow	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2			
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2		
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		j \rightarrow						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
i \downarrow	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0						
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
	1	A	0	1	←1	←1	←1	1
	2	C	0	1	←1	2	←2	←2
	3	G	0	1	←1	2	←2	←2
	4	A	0	1				
	5	C	0					

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2				
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2			
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2	3		
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$												
		0	1	2	3	4	5	6						
		<table border="1"> <tr> <td>A</td> <td>G</td> <td>C</td> <td>A</td> <td>T</td> <td>A</td> </tr> </table>							A	G	C	A	T	A
A	G	C	A	T	A									
$i \downarrow$	0	0	0	0	0	0	0	0						
1	A	0	1	1	1	1	1	1						
2	C	0	1	1	2	2	2	2						
3	G	0	1	2	2	2	2	2						
4	A	0	1	2	2	3	3							
5	C	0												

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2	3	3	3
5	C	0						

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2	3	3	3
5	C	0	1					

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2	3	3	3
5	C	0	1	2				

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2	3	3	3
5	C	0	1	2	3			

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2	3	3	3
5	C	0	1	2	3	3		

Longest Common Subsequence: Schritt 3

Rekurrenz

- $LCS(i, 0) = LCS(0, j) = 0$
- $LCS(i, j) = \begin{cases} LCS(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{LCS(i, j-1), LCS(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2	3	3	3
5	C	0	1	2	3	3	3	

Longest Common Subsequence: Schritt 3

Rekurrenz

- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

		$j \rightarrow$						
		0	1	2	3	4	5	6
			A	G	C	A	T	A
$i \downarrow$	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2
3	G	0	1	2	2	2	2	2
4	A	0	1	2	2	3	3	3
5	C	0	1	2	3	3	3	3

Longest Common Subsequence: Schritt 3

Rekurrenz

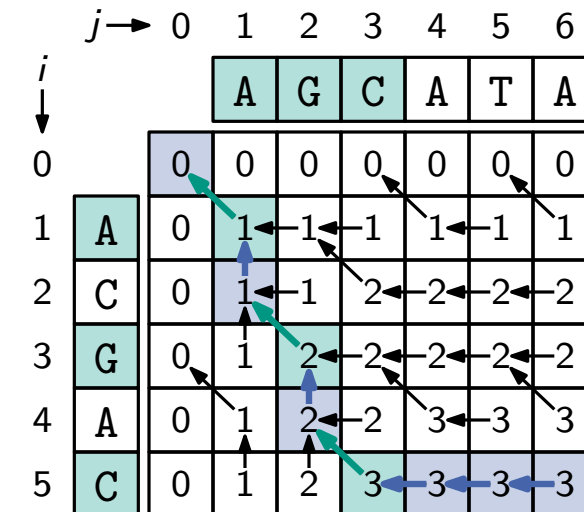
- $\text{LCS}(i, 0) = \text{LCS}(0, j) = 0$
- $\text{LCS}(i, j) = \begin{cases} \text{LCS}(i-1, j-1) + 1 & \text{wenn } a_i = b_j \\ \max\{\text{LCS}(i, j-1), \text{LCS}(i-1, j)\} & \text{wenn } a_i \neq b_j \end{cases}$

Antworten auf die Fragen von Schritt 3

- speichere für jedes $(i, j) \in [0, n] \times [0, m]$ eine Zahl
- \rightarrow Array von Arrays
- Eintrag bei (i, j) ggf. ab von Einträgen bei: $(i-1, j-1)$, $(i, j-1)$ und $(i-1, j)$
- von oben links nach unten rechts ausfüllen (Zeilenweise)
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt
 - Rückverfolgung dieser Pointer \rightarrow Lösung
(jeder diagonale Schritt ist gemeinsames Zeichen)

Schritt 3: Iterative Berechnung der Teillösungen




- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?



Subset Sum

Situation

- ihr braucht koeri + Pommes (3,70€)
- ihr wollt gerne passend bezahlen




	Reine Kalbsbratwurst mit Currysoße und Baguette ^[Se,Sn,We]	2,40 €
		☆☆☆
	Vegane Bratwurst mit Currysoße und Baguette ^[Se,Sn,So,We]	2,40 €
		★★★★
	koerifrites	1,30 €
		★★★★☆



Subset Sum

Situation

- ihr braucht koeri + Pommes (3,70€)
- ihr wollt gerne passend bezahlen




	Reine Kalbsbratwurst mit Currysoße und Baguette ^[Se,Sn,We]	2,40 €
		☆☆☆
	Vegane Bratwurst mit Currysoße und Baguette ^[Se,Sn,So,We]	2,40 €
		★★★★
	koerifrites	1,30 €
		★★★★



Subset Sum

Situation

- ihr braucht koeri + Pommes (3,70€)
- ihr wollt gerne passend bezahlen

	Reine Kalbsbratwurst mit Currysoße und Baguette ^[Se,Sn,We]	2,40 €
		☆☆☆
	Vegane Bratwurst mit Currysoße und Baguette ^[Se,Sn,So,We]	2,40 €
		☆☆☆
	koerifrites	1,30 €
		☆☆☆

Problem: Subset Sum




Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?



Subset Sum

Situation

- ihr braucht koeri + Pommes (3,70€)
- ihr wollt gerne passend bezahlen

	Reine Kalbsbratwurst mit Currysoße und Baguette ^[Se,Sn,We]	2,40 €
		☆☆☆
	Vegane Bratwurst mit Currysoße und Baguette ^[Se,Sn,So,We]	2,40 €
		☆☆☆
	koerifrites	1,30 €
		☆☆☆

Problem: Subset Sum

Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Anmerkungen

- das ist in gewissem Sinne ein schweres Problem
- mehr dazu in TGI nächstes Semester
- aber: S nicht zu groß \rightarrow effiziente Lösung mit DP



Subset Sum: Teilprobleme und Rekurrenz

Problem: Subset Sum Problem

Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Schritt 1: Spezifikation der Teilprobleme



Subset Sum: Teilprobleme und Rekurrenz

Problem: Subset Sum Problem

Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Schritt 1: Spezifikation der Teilprobleme

Welche Teilprobleme würdet ihr betrachten?

Subset Sum: Teilprobleme und Rekurrenz

Problem: Subset Sum Problem

Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Schritt 1: Spezifikation der Teilprobleme

- betrachte ein Paar (i, s) mit $i \leq n$ und $s \leq S$
- Gibt es Teilmenge $A' \subseteq \{a_1, \dots, a_i\}$, sodass $\sum_{a \in A'} a = s$?

Subset Sum: Teilprobleme und Rekurrenz

Problem: Subset Sum Problem

Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Schritt 1: Spezifikation der Teilprobleme

- betrachte ein Paar (i, s) mit $i \leq n$ und $s \leq S$
- Gibt es Teilmenge $A' \subseteq \{a_1, \dots, a_i\}$, sodass $\sum_{a \in A'} a = s$?
- $i = n$ und $s = S$ liefert Gesamtlösung

Subset Sum: Teilprobleme und Rekurrenz

Problem: Subset Sum Problem

Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Schritt 1: Spezifikation der Teilprobleme

- betrachte ein Paar (i, s) mit $i \leq n$ und $s \leq S$
- Gibt es Teilmenge $A' \subseteq \{a_1, \dots, a_i\}$, sodass $\sum_{a \in A'} a = s$?
- $i = n$ und $s = S$ liefert Gesamtlösung

Schritt 2: Aufstellung der Rekurrenz

- Ziel: **SSP** $(i, s) = \text{true}$ genau dann wenn es für das Paar (i, s) eine Lösung gibt

Subset Sum: Teilprobleme und Rekurrenz

Problem: Subset Sum Problem

Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Schritt 1: Spezifikation der Teilprobleme

- betrachte ein Paar (i, s) mit $i \leq n$ und $s \leq S$
- Gibt es Teilmenge $A' \subseteq \{a_1, \dots, a_i\}$, sodass $\sum_{a \in A'} a = s$?
- $i = n$ und $s = S$ liefert Gesamtlösung

Schritt 2: Aufstellung der Rekurrenz

- Ziel: $\text{SSP}(i, s) = \text{true}$ genau dann wenn es für das Paar (i, s) eine Lösung gibt
- $\text{SSP}(0, s) = \text{true} \Leftrightarrow s = 0$ (false sonst) und $\text{SSP}(i, s) = \text{false}$ für $s < 0$

Subset Sum: Teilprobleme und Rekurrenz

Problem: Subset Sum Problem

Gegeben eine Multimenge $A = \{a_1, \dots, a_n\}$ von natürlichen Zahlen und eine natürliche Zahl S . Gibt es eine Teilmenge $A' \subseteq A$, sodass $\sum_{a \in A'} a = S$?

Schritt 1: Spezifikation der Teilprobleme

- betrachte ein Paar (i, s) mit $i \leq n$ und $s \leq S$
- Gibt es Teilmenge $A' \subseteq \{a_1, \dots, a_i\}$, sodass $\sum_{a \in A'} a = s$?
- $i = n$ und $s = S$ liefert Gesamtlösung

Schritt 2: Aufstellung der Rekurrenz

- Ziel: $\text{SSP}(i, s) = \text{true}$ genau dann wenn es für das Paar (i, s) eine Lösung gibt
- $\text{SSP}(0, s) = \text{true} \Leftrightarrow s = 0$ (false sonst) und $\text{SSP}(i, s) = \text{false}$ für $s < 0$
- $\text{SSP}(i, s) = \text{SSP}(i-1, s) \vee \text{SSP}(i-1, s - a_i)$
 a_i nicht gewählt a_i gewählt

(Korrektheit folgt induktiv)

Subset Sum: Schritt 3

Rekurrenz

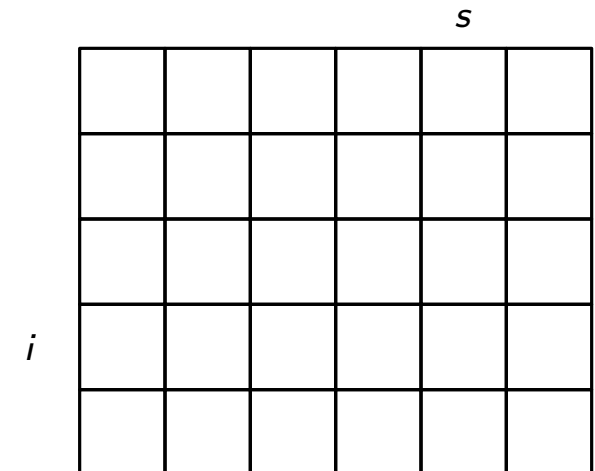
- $SSP(0, s) = \text{true} \Leftrightarrow s = 0$
- $SSP(i, s) = \text{false}$ für $s < 0$
- $SSP(i, s) = SSP(i - 1, s) \vee SSP(i - 1, s - a_i)$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- $SSP(i, s) = \text{false}$ für $s < 0$ brauchen wir nicht explizit speichern
- für jedes $(i, s) \in [0, n] \times [0, S]$ ein boolescher Wert \rightarrow Array von Arrays



Subset Sum: Schritt 3

Rekurrenz

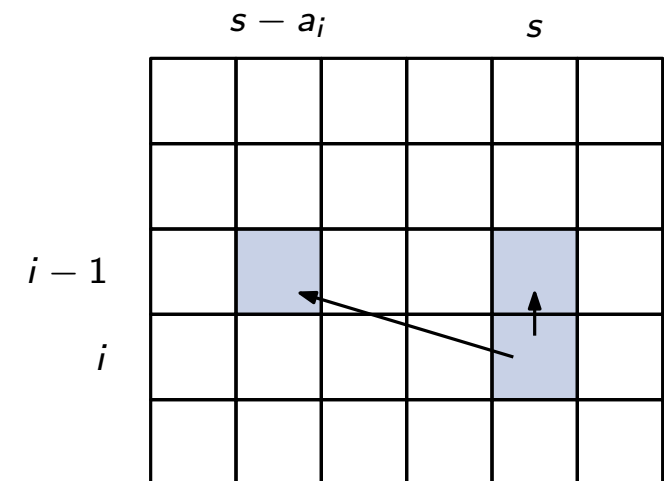
- $SSP(0, s) = \text{true} \Leftrightarrow s = 0$
- $SSP(i, s) = \text{false}$ für $s < 0$
- $SSP(i, s) = SSP(i - 1, s) \vee SSP(i - 1, s - a_i)$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- $SSP(i, s) = \text{false}$ für $s < 0$ brauchen wir nicht explizit speichern
- für jedes $(i, s) \in [0, n] \times [0, S]$ ein boolescher Wert \rightarrow Array von Arrays
- Einträge in Zeile i hängen von Einträgen in Zeile $i - 1$ ab
- Reihenfolge: Tabelle zeilenweise ausfüllen



Subset Sum: Schritt 3

Rekurrenz

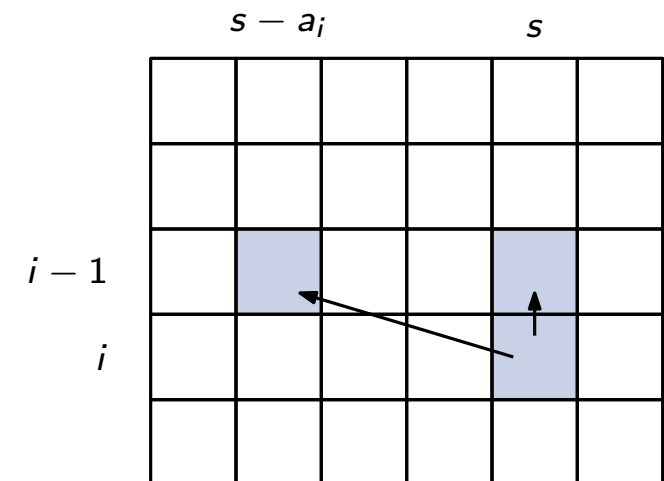
- $SSP(0, s) = \text{true} \Leftrightarrow s = 0$
- $SSP(i, s) = \text{false}$ für $s < 0$
- $SSP(i, s) = SSP(i - 1, s) \vee SSP(i - 1, s - a_i)$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- $SSP(i, s) = \text{false}$ für $s < 0$ brauchen wir nicht explizit speichern
- für jedes $(i, s) \in [0, n] \times [0, S]$ ein boolescher Wert \rightarrow Array von Arrays
- Einträge in Zeile i hängen von Einträgen in Zeile $i - 1$ ab
- Reihenfolge: Tabelle zeilenweise ausfüllen
- zur Rekonstruktion der Lösung:
 - merke für jeden Eintrag, von wo er kommt
 - Rückverfolgung dieser Pointer \rightarrow Lösung
(jeder diagonale Schritt entspricht ausgewähltem Element)

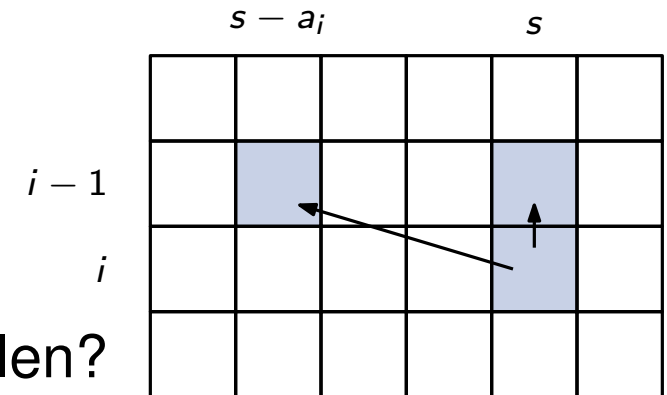


Alternativer Sichtweise für Schritt 3: Pfadfindung

Rekurrenz

- $SSP(0, s) = \text{true} \Leftrightarrow s = 0$
- $SSP(i, s) = \text{false}$ für $s < 0$
- $SSP(i, s) = SSP(i-1, s) \vee SSP(i-1, s - a_i)$

Bisher: Welche Einträge muss ich anschauen, um (i, s) auszufüllen?

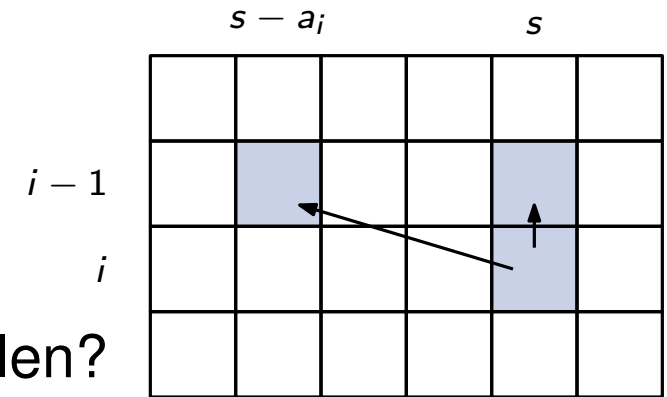


Alternativer Sichtweise für Schritt 3: Pfadfindung

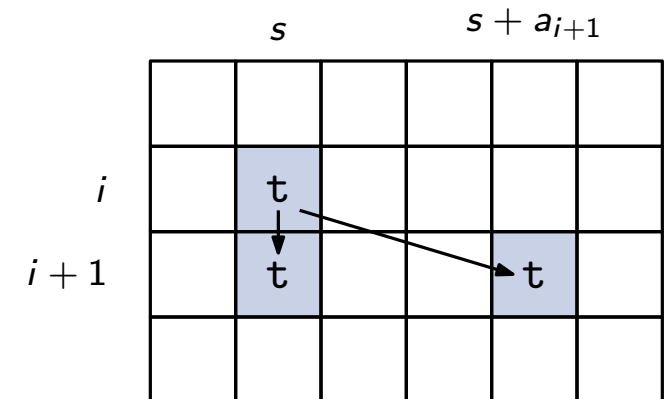
Rekurrenz

- $SSP(0, s) = \text{true} \Leftrightarrow s = 0$
- $SSP(i, s) = \text{false}$ für $s < 0$
- $SSP(i, s) = SSP(i-1, s) \vee SSP(i-1, s - a_i)$

Bisher: Welche Einträge muss ich anschauen, um (i, s) auszufüllen?



Jetzt: Welche Einträge kann ich auf `true` setzen, wenn ich weiß, dass $SSP(i, s) = \text{true}$?

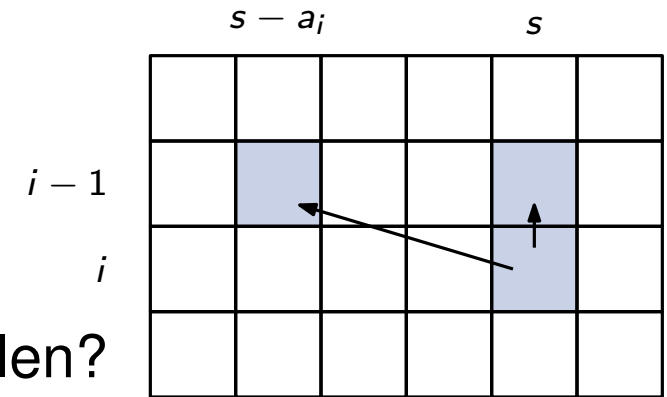


Alternativer Sichtweise für Schritt 3: Pfadfindung

Rekurrenz

- $SSP(0, s) = \text{true} \Leftrightarrow s = 0$
- $SSP(i, s) = \text{false}$ für $s < 0$
- $SSP(i, s) = SSP(i-1, s) \vee SSP(i-1, s - a_i)$

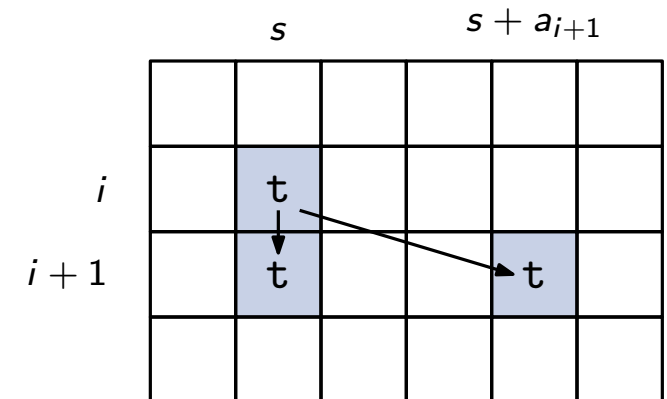
Bisher: Welche Einträge muss ich anschauen, um (i, s) auszufüllen?



Jetzt: Welche Einträge kann ich auf true setzen, wenn ich weiß, dass $SSP(i, s) = \text{true}$?

Pfadfindung in einem Graphen

- fasse jedes Paar (i, s) als Knoten auf
- Kanten: $(i, s) \rightarrow (i+1, s)$ und $(i, s) \rightarrow (i+1, s + a_{i+1})$
- Ja-Instanz \Leftrightarrow es gibt einen Pfad von $(0, 0)$ zu (n, S)

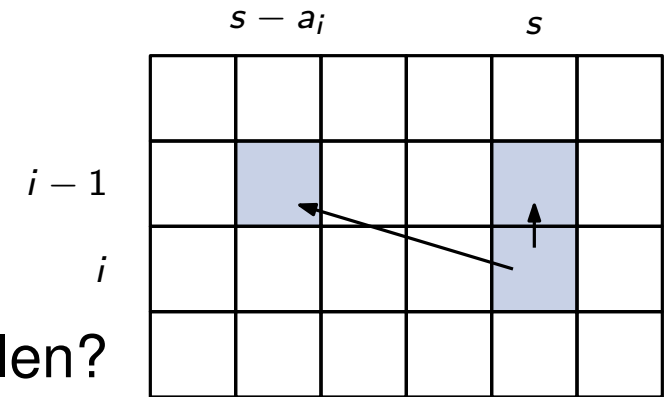


Alternativer Sichtweise für Schritt 3: Pfadfindung

Rekurrenz

- $SSP(0, s) = \text{true} \Leftrightarrow s = 0$
- $SSP(i, s) = \text{false}$ für $s < 0$
- $SSP(i, s) = SSP(i-1, s) \vee SSP(i-1, s - a_i)$

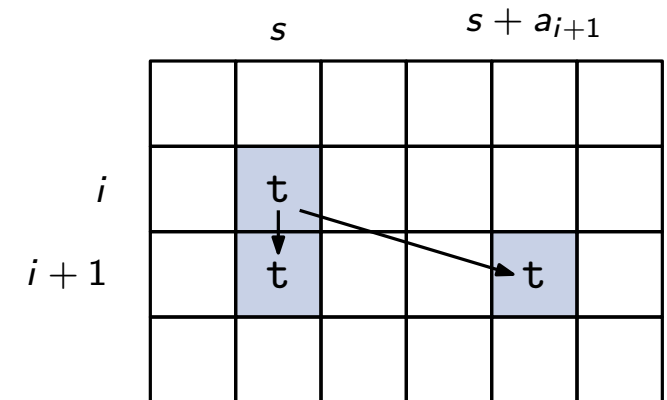
Bisher: Welche Einträge muss ich anschauen, um (i, s) auszufüllen?



Jetzt: Welche Einträge kann ich auf true setzen, wenn ich weiß, dass $SSP(i, s) = \text{true}$?

Pfadfindung in einem Graphen

- fasse jedes Paar (i, s) als Knoten auf
- Kanten: $(i, s) \rightarrow (i+1, s)$ und $(i, s) \rightarrow (i+1, s + a_{i+1})$
- Ja-Instanz \Leftrightarrow es gibt einen Pfad von $(0, 0)$ zu (n, S)
- beachte: Graph nur implizit repräsentiert
- Vorteil: wir besuchen in der Tabelle nur Zellen, die true sind



(ggf. deutlich weniger als alle Zellen)

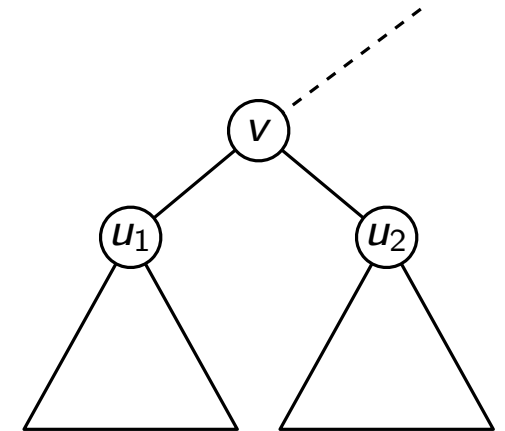
Längster Pfad im Baum

Problem: Longest Path

Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur



Erinnerung: 1. Spezifikation der Teilprobleme

- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?

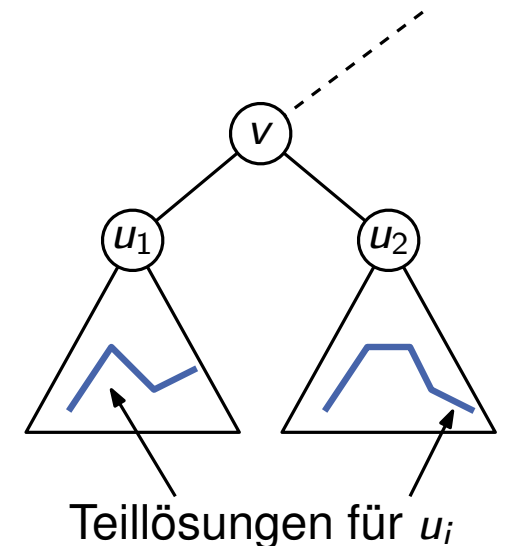
Längster Pfad im Baum

Problem: Longest Path

Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung für v : längster Pfad in Teilbaum T_v unter v



Erinnerung: 1. Spezifikation der Teilprobleme

- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?

Längster Pfad im Baum

Problem: Longest Path

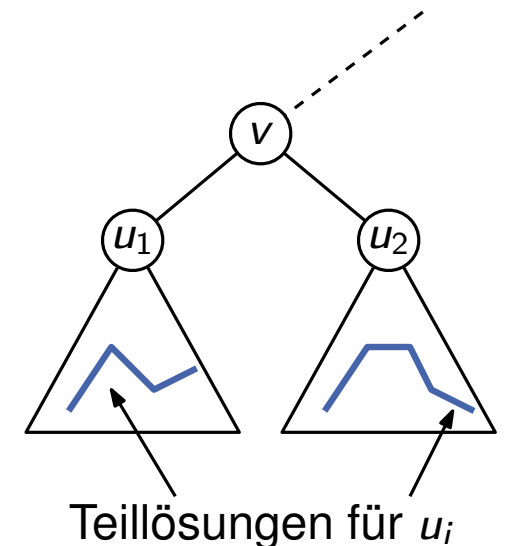
Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung für v : längster Pfad in Teilbaum T_v unter v

Schritt 2: Aufstellung der Rekurrenz

- erhalte Lösung für v aus Lösung für Kinder u_1, \dots, u_k



Erinnerung: 1. Spezifikation der Teilprobleme

- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?

Längster Pfad im Baum

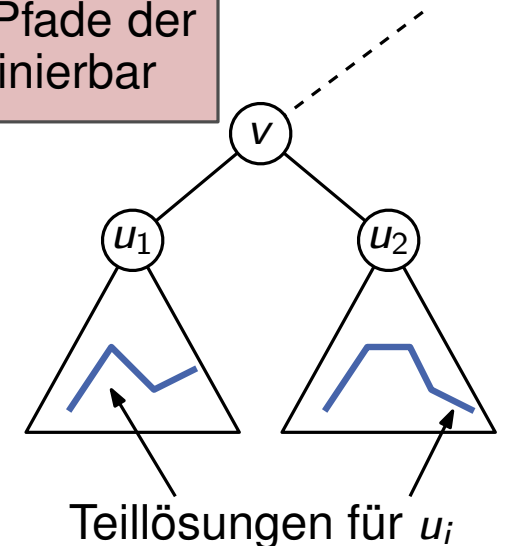
Problem: Longest Path

Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung für v : längster Pfad in Teilbaum T_v unter v

Problem: längste Pfade der T_{u_i} ggf. nicht kombinierbar



Schritt 2: Aufstellung der Rekurrenz

- erhalte Lösung für v aus Lösung für Kinder u_1, \dots, u_k

Erinnerung: 1. Spezifikation der Teilprobleme

- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?

Längster Pfad im Baum

Problem: Longest Path

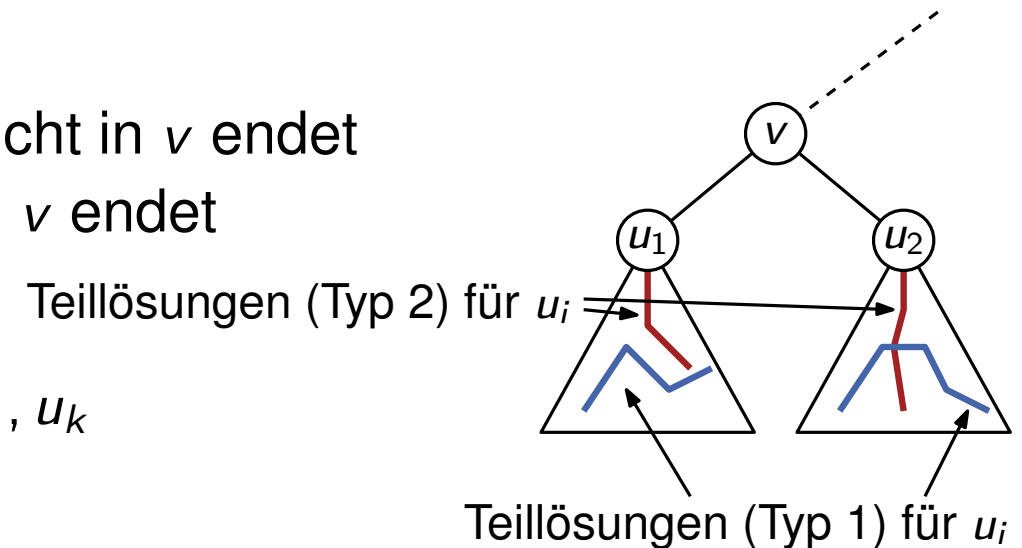
Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung (Typ 1) für v : längster Pfad in T_v , der nicht in v endet
- Teillösung (Typ 2) für v : längster Pfad in T_v , der in v endet

Schritt 2: Aufstellung der Rekurrenz

- erhalte Lösung für v aus Lösung für Kinder u_1, \dots, u_k



Erinnerung: 1. Spezifikation der Teilprobleme

- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?

Längster Pfad im Baum

Problem: Longest Path

Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

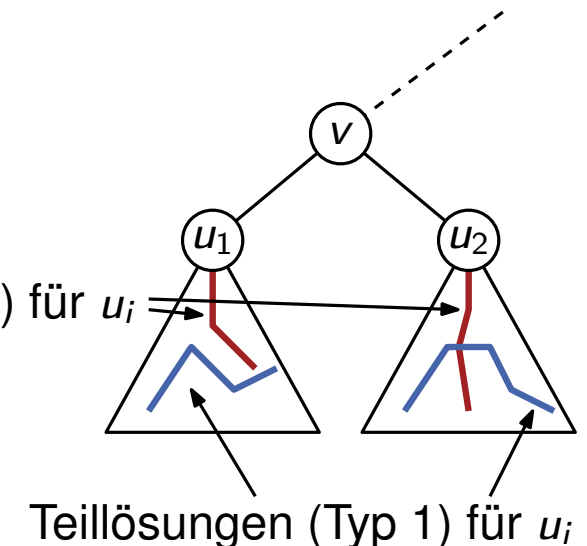
Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung (Typ 1) für v : längster Pfad in T_v , der nicht in v endet
- Teillösung (Typ 2) für v : längster Pfad in T_v , der in v endet

Schritt 2: Aufstellung der Rekurrenz

- erhalte Lösung für v aus Lösung für Kinder u_1, \dots, u_k
- Ziel: $LP_t(v)$ repräsentiert Teillösung von Typ t für v

Teillösungen (Typ 2) für u_i



Längster Pfad im Baum

Problem: Longest Path

Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

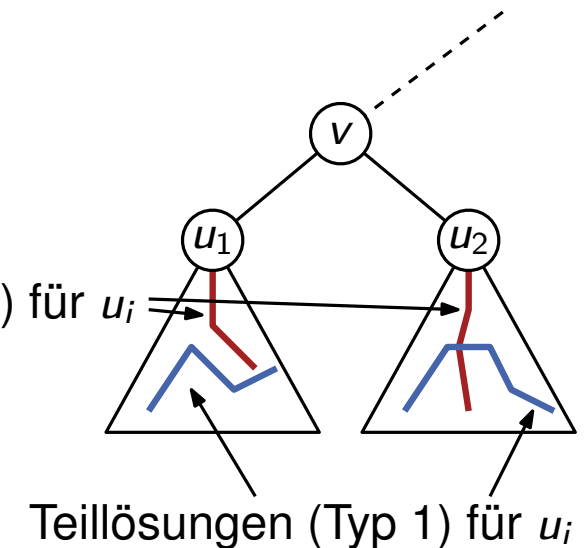
Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung (Typ 1) für v : längster Pfad in T_v , der nicht in v endet
- Teillösung (Typ 2) für v : längster Pfad in T_v , der in v endet

Schritt 2: Aufstellung der Rekurrenz

- erhalte Lösung für v aus Lösung für Kinder u_1, \dots, u_k
- Ziel: $LP_t(v)$ repräsentiert Teillösung von Typ t für v
- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$

Teillösungen (Typ 2) für u_i



Längster Pfad im Baum

Problem: Longest Path

Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

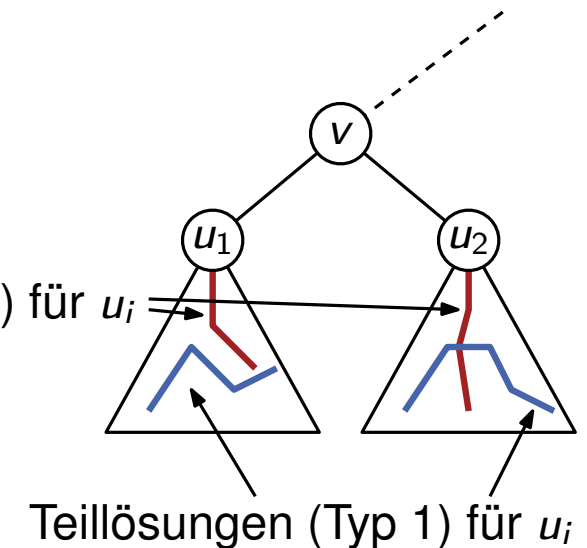
Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung (Typ 1) für v : längster Pfad in T_v , der nicht in v endet
- Teillösung (Typ 2) für v : längster Pfad in T_v , der in v endet

Schritt 2: Aufstellung der Rekurrenz

- erhalte Lösung für v aus Lösung für Kinder u_1, \dots, u_k
- Ziel: $\mathbf{LP}_t(v)$ repräsentiert Teillösung von Typ t für v
- Falls v ein Blatt: $\mathbf{LP}_1(v) = \mathbf{LP}_2(v) = 0$
- $\mathbf{LP}_2(v) = \max_i \{ \mathbf{LP}_2(u_i) + 1 \}$

Teillösungen (Typ 2) für u_i



Längster Pfad im Baum

Problem: Longest Path

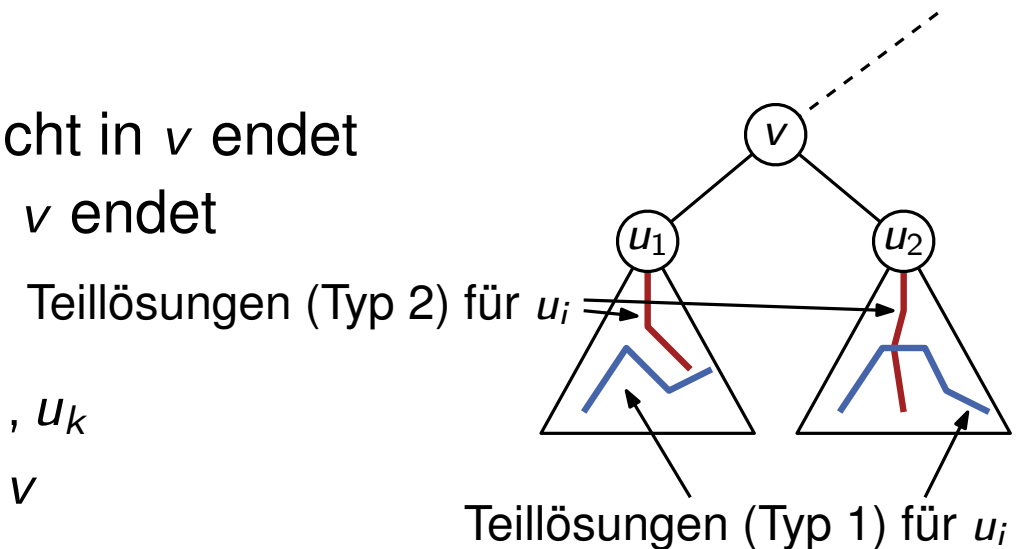
Gegeben einen (gewurzelten) Baum T . Wie lang ist der längste Pfad in T ?

Schritt 1: Spezifikation der Teilprobleme

- strukturelle Induktion über Baumstruktur
- Teillösung (Typ 1) für v : längster Pfad in T_v , der nicht in v endet
- Teillösung (Typ 2) für v : längster Pfad in T_v , der in v endet

Schritt 2: Aufstellung der Rekurrenz

- erhalte Lösung für v aus Lösung für Kinder u_1, \dots, u_k
- Ziel: $\mathbf{LP}_t(v)$ repräsentiert Teillösung von Typ t für v
- Falls v ein Blatt: $\mathbf{LP}_1(v) = \mathbf{LP}_2(v) = 0$
- $\mathbf{LP}_2(v) = \max_i \{ \mathbf{LP}_2(u_i) + 1 \}$
- $\mathbf{LP}_1(v) = \max \left[\max_i \{ \max \{ \mathbf{LP}_1(u_i), \mathbf{LP}_2(u_i) \} \}, \max_{i \neq j} \{ \mathbf{LP}_2(u_i) + \mathbf{LP}_2(u_j) + 2 \} \right]$



Längster Pfad im Baum: Korrektheit

Rekurrenz

- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

Was wollen wir zeigen?

- Ziel: $LP_i(v)$ repräsentiert Teillösung von Typ i für v
- Typ 1: längster Pfad in Teilbaum T_v unter v
- Typ 2: längster Pfad in T_v , der in v endet

Längster Pfad im Baum: Korrektheit

Rekurrenz

- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

Was wollen wir zeigen?

- Ziel: $LP_i(v)$ repräsentiert Teillösung von Typ i für v
- Typ 1: längster Pfad in Teilbaum T_v unter v
- Typ 2: längster Pfad in T_v , der in v endet

Korrektheit mittels struktureller Induktion

- korrekt für die Blätter
- korrekt für v falls korrekt für die Kinder u_1, \dots, u_k

Längster Pfad im Baum: Korrektheit

Rekurrenz

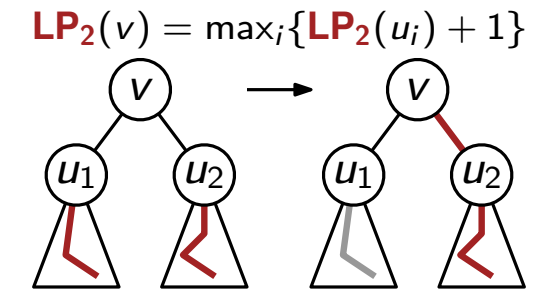
- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

Was wollen wir zeigen?

- Ziel: $LP_i(v)$ repräsentiert Teillösung von Typ i für v
- **Typ 1**: längster Pfad in Teilbaum T_v unter v
- **Typ 2**: längster Pfad in T_v , der in v endet

Korrektheit mittels struktureller Induktion

- korrekt für die Blätter
- korrekt für v falls korrekt für die Kinder u_1, \dots, u_k
- **Typ 2**: Pfad führt über ein Kind u_i (+1 wegen $\{v, u_i\}$)



Längster Pfad im Baum: Korrektheit

Rekurrenz

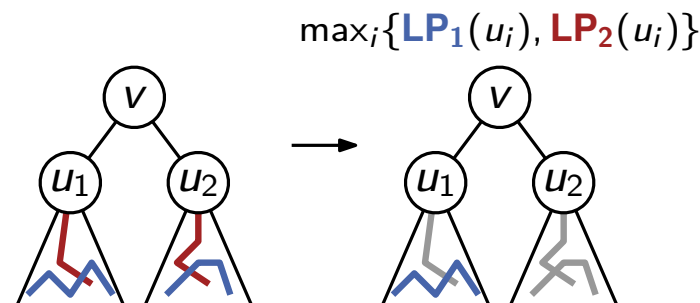
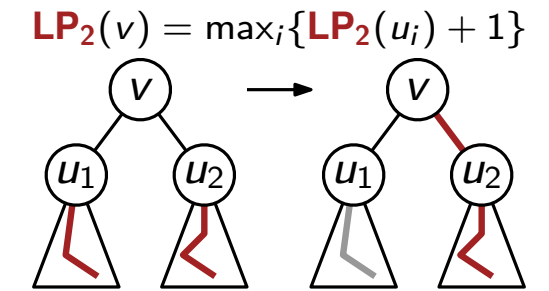
- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

Was wollen wir zeigen?

- Ziel: $LP_i(v)$ repräsentiert Teillösung von Typ i für v
- Typ 1: längster Pfad in Teilbaum T_v unter v
- Typ 2: längster Pfad in T_v , der in v endet

Korrektheit mittels struktureller Induktion

- korrekt für die Blätter
- korrekt für v falls korrekt für die Kinder u_1, \dots, u_k
- Typ 2:** Pfad führt über ein Kind u_i (+1 wegen $\{v, u_i\}$)
- Typ 1, Fall 1:** längster Pfad in T_v enthält v nicht



Längster Pfad im Baum: Korrektheit

Rekurrenz

- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

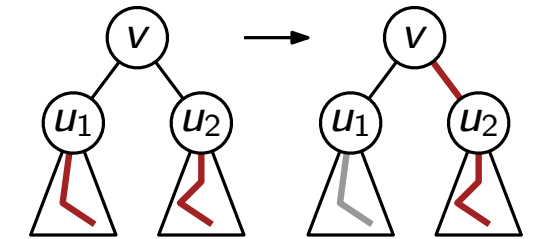
Was wollen wir zeigen?

- Ziel: $LP_i(v)$ repräsentiert Teillösung von Typ i für v
- Typ 1: längster Pfad in Teilbaum T_v unter v
- Typ 2: längster Pfad in T_v , der in v endet

Korrektheit mittels struktureller Induktion

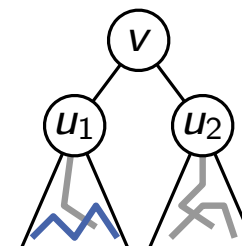
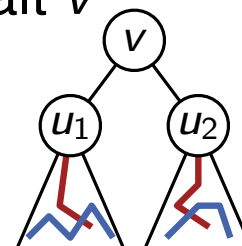
- korrekt für die Blätter
- korrekt für v falls korrekt für die Kinder u_1, \dots, u_k
- Typ 2:** Pfad führt über ein Kind u_i (+1 wegen $\{v, u_i\}$)
- Typ 1, Fall 1:** längster Pfad in T_v enthält v nicht
- Typ 1, Fall 2:** längster Pfad in T_v enthält v

$$LP_2(v) = \max_i \{LP_2(u_i) + 1\}$$

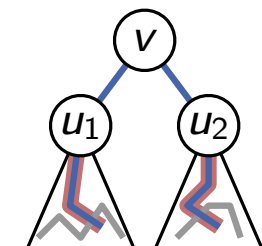


$$\max_i \{LP_1(u_i), LP_2(u_i)\}$$

$$\max_{i \neq j} \{LP_2(u_i) + LP_2(u_j) + 2\}$$



oder



Längster Pfad im Baum: Schritt 3

Rekurrenz

- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- speichere zwei Werte an jedem Knoten v : $LP_1(v)$ und $LP_2(v)$

Längster Pfad im Baum: Schritt 3

Rekurrenz

- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- speichere zwei Werte an jedem Knoten v : $LP_1(v)$ und $LP_2(v)$
- Eltern hängen von Kindern ab \rightarrow Reihenfolge: bottom-up (z.B. entsprechend der Lagen im BFS-Baum)

Längster Pfad im Baum: Schritt 3

Rekurrenz

- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- speichere zwei Werte an jedem Knoten v : $LP_1(v)$ und $LP_2(v)$
- Eltern hängen von Kindern ab \rightarrow Reihenfolge: bottom-up (z.B. entsprechend der Lagen im BFS-Baum)
- tatsächliche Lösung berechnen: wie bei vorherigen DPs

Längster Pfad im Baum: Schritt 3

Rekurrenz

- Falls v ein Blatt: $LP_1(v) = LP_2(v) = 0$
- $LP_2(v) = \max_i \{LP_2(u_i) + 1\}$
- $LP_1(v) = \max \left[\max_i \{ \max \{ LP_1(u_i), LP_2(u_i) \} \}, \max_{i \neq j} \{ LP_2(u_i) + LP_2(u_j) + 2 \} \right]$

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

Antworten auf die Fragen von Schritt 3

- speichere zwei Werte an jedem Knoten v : $LP_1(v)$ und $LP_2(v)$
- Eltern hängen von Kindern ab \rightarrow Reihenfolge: bottom-up (z.B. entsprechend der Lagen im BFS-Baum)
- tatsächliche Lösung berechnen: wie bei vorherigen DPs

Anmerkung

- Teillösungen überlappen nicht: Teillösung jedes Kinds u_i nur für den einen Elter v relevant
- rekursive Implementierung in dem Fall also auch ok (entspricht im Prinzip einer DFS)

In drei Schritten zum dynamischen Programm

Schritt 1: Spezifikation der Teilprobleme

- Welche Zwischenergebnisse wollen wir berechnen?
- Worüber geht die Induktion?
- Induktion über Parameter oder eine Baumstruktur

schwer
erfordert Kreativität
ggf. richtige Idee nötig
interagiert mit Schritt 2

Schritt 2: Aufstellung der Rekurrenz für Teillösungen

- Wie berechnen wir neue aus alten Zwischenergebnissen?
- Korrektheit: vollständige oder strukturelle Induktion

nicht superschwer
(vorausgesetzt ihr habt eine gute
Spezifikation der Teilergebnisse)

Schritt 3: Iterative Berechnung der Teillösungen

- Wie verwalten wir die Zwischenergebnisse?
- Wie hängen die Zwischenergebnisse voneinander ab?
- In welcher Reihenfolge berechnen wir Zwischenergebnisse?
- Wie bekommen wir die tatsächliche Lösung?

leicht
(mit etwas Übung)

Zusammenfassung

DPs haben wir schon häufiger gesehen (ohne es zu wissen)

- Bellman–Ford
 - DP über die Anzahl Kanten aus denen die Pfade bestehen
 - Iteration i : berechne kürzeste Pfade unter allen Pfaden, die aus $\leq i$ Kanten bestehen

Zusammenfassung

DPs haben wir schon häufiger gesehen (ohne es zu wissen)

- Bellman–Ford
 - DP über die Anzahl Kanten aus denen die Pfade bestehen
 - Iteration i : berechne kürzeste Pfade unter allen Pfaden, die aus $\leq i$ Kanten bestehen
- Blatt 10, Aufgabe 3 – Handtuch nicht vergessen!
 - berechne längsten Pfad in einem DAG
 - DP über topologische Sortierung der Knoten

Zusammenfassung

DPs haben wir schon häufiger gesehen (ohne es zu wissen)

- Bellman–Ford
 - DP über die Anzahl Kanten aus denen die Pfade bestehen
 - Iteration i : berechne kürzeste Pfade unter allen Pfaden, die aus $\leq i$ Kanten bestehen
- Blatt 10, Aufgabe 3 – Handtuch nicht vergessen!
 - berechne längsten Pfad in einem DAG
 - DP über topologische Sortierung der Knoten

Lernziel

- ihr habt verstanden, wie DPs funktionieren \rightarrow schnelleres Verständnis unbekannter DPs
- ihr könnt zumindest einfache DPs selber bauen
(insbesondere, wenn die Teillösungen gegeben sind)