ETH zürich

Übung 2

Datenstrukturen und Algorithmen, D-MATH, ETH Zurich

Programm von heute

Feedback letzte Übung

C++ Container Bibliothek

Templates Wiederholung

Wiederholung Theorie
- Induktion

Use Case
- Subarray Sum Problem

Landau-Notation

- Geben Sie eine korrekte, kompakte Definition der Menge $\Theta(f)$ analog zur Definition der Mengen $\mathcal{O}(f)$ und $\Omega(f)$.

Landau-Notation

- Geben Sie eine korrekte, kompakte Definition der Menge $\Theta(f)$ analog zur Definition der Mengen $\mathcal{O}(f)$ und $\Omega(f)$.
- $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists a > 0,\ b > 0,\ n_0 \in \mathbb{N} : a \cdot f(n) \leq g(n) \leq b \cdot f(n)\ \forall n \geq n_0\}$

Landau-Notation

- Geben Sie eine korrekte, kompakte Definition der Menge $\Theta(f)$ analog zur Definition der Mengen $\mathcal{O}(f)$ und $\Omega(f)$.
- $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists a > 0,\ b > 0,\ n_0 \in \mathbb{N} : a \cdot f(n) \leq g(n) \leq b \cdot f(n)\ \forall n \geq n_0\}$
- $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists c > 0,\ n_0 \in \mathbb{N} : \frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n)\ \forall n \geq n_0\}$

Landau-Notation

Beweisen oder widerlegen Sie, mit $f, g : \mathbb{N} \to \mathbb{R}^+$.

(a) $f \in \mathcal{O}(g)$ genau dann wenn $g \in \Omega(f)$.

(e) $\log_a(n) \in \Theta(\log_b(n))$ für alle Konstanten $a, b \in \mathbb{N} \setminus \{1\}$

(g) Falls $f_1, f_2 \in \mathcal{O}(g)$ und $f(n) := f_1(n) \cdot f_2(n)$, dann gilt $f \in \mathcal{O}(g)$.

Landau-Notation

Funktionen sortieren: wenn Funktion f links der Funktion g steht, dann $f \in \mathcal{O}(g)$.

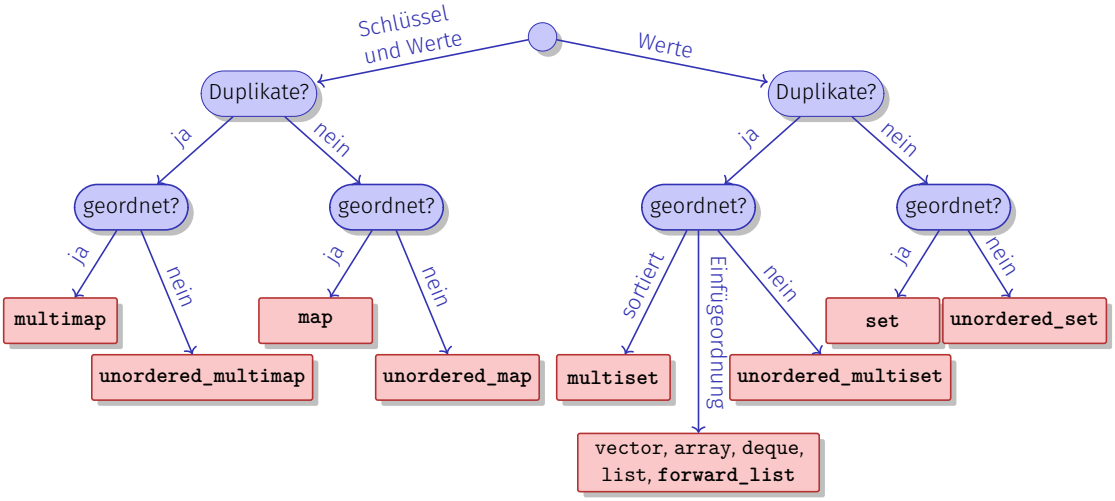
$2^{16},\ \log(n^4),\ \log^8(n),\ \sqrt{n},\ n \log n,\ \binom{n}{3},\ n^5 + n,\ \frac{2^n}{n^2},\ n!,\ n^n.$

Summe von Elementen in zweidimensionalem Feld

Probleme / Fragen?

2. C++ Container Bibliothek

C++ Container


Schlüssel und Werte
Werte
Duplikate?
Duplikate?
ja
nein
ja
nein
geordnet?
geordnet?
geordnet?
geordnet?
ja
nein
ja
nein
sortiert
Einfügeordnung
nein
ja
nein
multimap
unordered_multimap
map
unordered_map
multiset
vector, array, deque, list, forward_list
unordered_multiset
set
unordered_set

Sequenz-Container

`vector`	`array`	`deque`	`list`	`forward_list`
Zusammenh. dynamischer Speicher	Zusammenh. statischer Speicher	Nicht zush. dyn. Speicher	Nicht zush. dyn. Speicher	Nicht zush. dyn. Speicher
Wahlfreier Zugriff	Wahlfreier Zugriff	Wahlfreier Zugriff		
schnelles push/pop back		schnelles push/pop front/back	schnelles push/pop front/back	schnelles push/pop front
bid. Iteration	bid. Iteration	bid. Iteration	bid. Iteration	Iteration vorwärts

Mengen und Multimengen

- **`std::set<E>`** enthält jedes Element maximal einmal
- **`std::multiset<E>`** kann Elemente mehrfach enthalten
  - Iteration liefert die Elemente in absteigender Anordnung (in nicht-deterministischer Anordnung, wenn **`unordered_multiset`**)
  - **`std::multiset<E>::count(elem)`** liefert die Anzahl Vorkommnisse eines Elements

Mengen und Multimengen

- **`std::set<E>`** enthält jedes Element maximal einmal
- **`std::multiset<E>`** kann Elemente mehrfach enthalten
  - Iteration liefert die Elemente in absteigender Anordnung (in nicht-deterministischer Anordnung, wenn **`unordered_multiset`**)
  - **`std::multiset<E>::count(elem)`** liefert die Anzahl Vorkommnisse eines Elements

Beispiel von `std::multiset`

```
Content: Xanten Xenon Xenon Xenon Xerografie Xerophil Xylose
count("Xenon") = 3
count("Xylose") = 1
```

Zuordnungen

- **`std::map<K,V>`** enthält (Schlüssel-Wert) Paare, wobei ein Schlüssel maximal einem Element zugeordnet ist
- **`std::multimap<K,V>`** erlaubt mehrfache Zuordnungen
  - Iteration liefert die Elemente in absteigender Schlüsselanordnung (in nicht-deterministischer Anordnung, wenn **`unordered_multimap`**)
  - **`std::multimap<K,V>::count(key)`** gibt die Anzahl Vorkommnisse eines Schlüssels zurück
  - **`std::multimap<K,V>::equal_range(key)`** liefert alle Werte (in nichtdeterministischer Anordnung) für den gegebenen Schlüssel

Zuordnungen

- **`std::map<K,V>`** enthält (Schlüssel-Wert) Paare, wobei ein Schlüssel maximal einem Element zugeordnet ist
- **`std::multimap<K,V>`** erlaubt mehrfache Zuordnungen
  - Iteration liefert die Elemente in absteigender Schlüsselanordnung (in nicht-deterministischer Anordnung, wenn **`unordered_multimap`**)
  - **`std::multimap<K,V>::count(key)`** gibt die Anzahl Vorkommnisse eines Schlüssels zurück
  - **`std::multimap<K,V>::equal_range(key)`** liefert alle Werte (in nichtdeterministischer Anordnung) für den gegebenen Schlüssel

Beispiel von **`std::multimap<K,V>`**

```
Content: {2, er} {2, du} {2, es} {3, Axt} {3, sie} {4, Igel}
count(2) = 3
Values for key 2: er du es
```

3. Templates Wiederholung

Motivation

Ziel: generischer binärer Baum, ohne Codeduplikation

```
class Node { ... }; // Node of a binary search tree
auto n1 = Node<int>(5);
auto n2 = Node<std::string>("Zürich");
n1.insert(1);
n2.contains(2); // Compiler error
```

Motivation

Ziel: generischer binärer Baum, ohne Codeduplikation

```
class Node { ... }; // Node of a binary search tree
auto n1 = Node<int>(5);
auto n2 = Node<std::string>("Zürich");
n1.insert(1);
n2.contains(2); // Compiler error
```

Idee:

- Mache Klassen parametrisch in ihren Typen (=template Parameter)
- … so wie sie ja schon parametrisch in Werten sind (=Funktionsparameter)

Typen als Template Parameter

1. Ersetze in der konkreten Implementation einer Klasse den Typ, der generisch werden soll (z.B. **`int`**) durch einen Stellvertreter, z.B. **`T`**.
2. Stelle der Klasse das Konstrukt **`template<typename T>`** voran (ersetze **`T`** ggfs. durch den Stellvertreter)..

Das Konstrukt **`template<typename T>`** kann gelesen werden als **“für alle Typen `T`”**.

Klassentemplate

```
template <typename K>
class Node {
  K key;
  Node* left, right;
public:
  Node(K k, Node* l, Node* r): key(k), left(l), right(r) {}

  bool contains(K search_key) const {
    return search_key != key
      || left != nullptr && left->contains(search_key)
      || right != nullptr && right->contains(search_key)
  }
  ...
};
```

Funktionentemplate: analog

1. Ersetze in der konkreten Implementation einer Funktion den Typ, der generisch werden soll durch einen Namen, z.B. `T`,
2. Stelle der Funktion das Konstrukt `template<typename T>` voran (ersetze `T` ggfs. durch den gewählten Namen).

Beispiele

- Für freie Funktionen

```
template <typename T>
void swap(T& x, T& y) {
  T temp = x;
  x = y;
  y = temp;
}
```

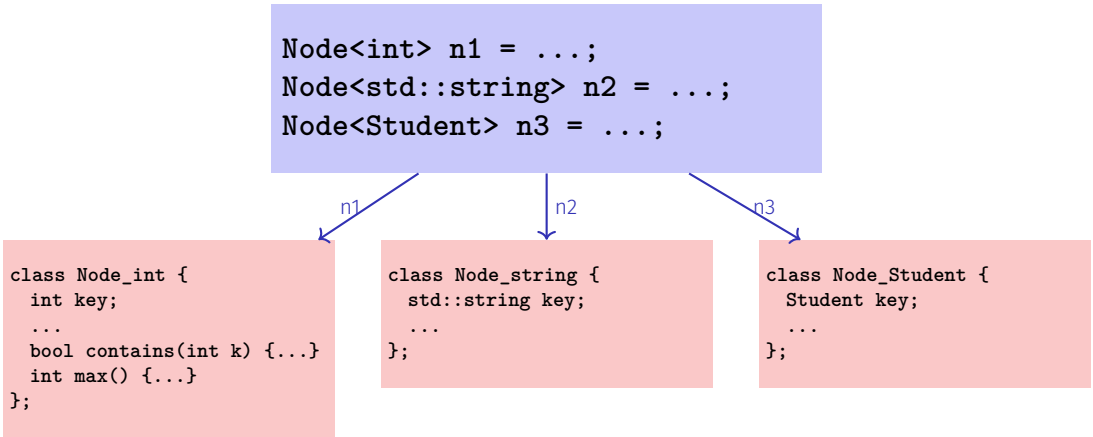
```
template <typename Iter>
void is_sorted(Iter begin, Iter end){
  ...
}
```

- Für Operatoren

```
template <typename T>
ostream& operator<<(ostream& out, const Node<T> root) {
  ...
}
```

Semantik (Code-Generation)

Für jede Instanz eines templates generiert der Compiler eine entsprechende Klasse (oder Funktion) → statische Codegenerierung



Semantik (Code-Generation)

Für jede Instanz eines templates generiert der Compiler eine entsprechende Klasse (oder Funktion) $\rightarrow$ statische Codegenerierung

Frage: Was bedeutet das für die separate Kompilation

- Sollten templates in Deklarations- (.h) und Definitionsfiles (.cpp) aufgeteilt werden?
- Ist es möglich binäre vorkompilierte Files mit den Header-Files bereitzustellen?

Typenprüfung

- Templates: syntaktische Prüfung
- Instanzen: übliche Prüfung

```
template <typename T>
T abs(T v) {
  return 0 <= v ? v : -v;
}
// main
foo(8); // OK
```

```
template <typename T>
void swap(T& x, T& y) {
  ...
}
// main
double a = 1.0;
double b = 7;
swap(a, b); // OK
```

```
emplate <typename T>
T abs(T v) {
  return 0 <= v ? v : -v; // Error
}
// main
foo("hi"); // Error
```

```
template <typename T>
void swap(T& x, T& y) {
  ...
}
// main
double a = 1.0;
string b = "seven";
swap(a, b); // Error
```

Andere Sprachen

Alle Sprachen versuchen Code-Wiederverwendung zu ermöglichen.

- C++, Rust:
  - Statische Codeerzeugung
  - Kein Laufzeitoverhead
  - schwierig mit OOP zu vereinbaren
- C#, Scala (, Java)
  - Typ-Parameter werden zu Laufzeitwerten
  - unterstützt OOP gut
  - kleiner Laufzeitoverhead
- Python, JavaScript:
  - dynamische Typisierung (duck typing)
  - kein syntaktischer Overhead
  - signifikanter Laufzeitoverhead

3.1 auto vs templates

auto

- Platzhalter für einen Typ
  - Typ muss aus dem unmittelbaren Kontext abgeleitet werden können: Initialisierung oder return
  - Benutzer könnte den Typ hinschreiben, überlässt es aber dem Compiler

```
std::vector<int> vec = ...;
auto it = vec.cbegin();
// placeholder for td::vector<int>::const_iterator
```

- Beispiele, wo das schiefgeht

```
auto x; // x has no initializer
x = 0.0;
auto first_or_else(std::vector<int> data, unsigned int or_else) {
  if (data.size() == 0) return or_else;
  else return data[0];
}
```

Templates

- Parameter sind unbekannt, bis instanziert wird

```
template <typename N>
char sign(N v) {
  if (0 <= v) return '+';
  else return '-';
}

template <typename T1, typename T2>
struct Pair {
  T1 fst;
  T2 snd;
};
```

- Instanziierung kann überall passieren

```
Pair<int, double> p1 = Pair{1, 0.1};
auto p2 = Pair<std::string, bool>{"Brazil", true};
```

Templates und auto kombinieren

`auto` im Template muss nach der Instanziierung festgelegt werden

```
template <typename C>
void print(C container) {
  for (auto& e : container)
  std::cout << e << ' ';
}

std::vector<int> numbers = {1, 2, 3};
print(numbers); // now auto can be determined

std::vector<std::string> airports = {"LAX", "LDN", "ZHR"};
print(airports); // now auto can be determined
```

Templates und auto kombinieren

`auto` im Template muss nach der Instanziierung festgelegt werden

```
template <typename C>
void print(C container) {
  for (auto& e : container)
  std::cout << e << ' ';
}
```

Frage: Ist es möglich `auto` in diesem Beispiel nicht zu verwenden?

Templates und auto kombinieren

`auto` im Template muss nach der Instanziierung festgelegt werden

```
template <typename C>
void print(C container) {
  for (auto& e : container)
  std::cout << e << ' ';
}
```

Frage: Ist es möglich `auto` in diesem Beispiel nicht zu verwenden?

Antwort: Ja, zum Beispiel indem man `auto` durch einen zusätzlichen Template-Parameter `E` ersetzt

Von auto zu Templates

- Vor **C++20** sind auto-Funktionsparameter verboten

```
void print(auto x) {...} // Compiler error
```

Von auto zu Templates

- Vor **C++20** sind auto-Funktionsparameter verboten

```
void print(auto x) {...} // Compiler error
```

Frage: Was meinen Sie, warum?

Von auto zu Templates

- Vor **C++20** sind auto-Funktionsparameter verboten

```
void print(auto x) {...} // Compiler error
```

Frage: Was meinen Sie, warum?
Antwort: Der Typ kann nicht vom Kontext abgeleitet werden

Von auto zu Templates

- Vor C++20 sind auto-Funktionsparameter verboten

```
void print(auto x) {...} // Compiler error
```

Frage: Was meinen Sie, warum?
Antwort: Der Typ kann nicht vom Kontext abgeleitet werden

- Seit C++20 sind auto-Funktionsparameter erlaubt

```
void print(auto x) {...} // ok
```

Es ist natürlich immer noch nicht möglich, herauszufinden, wofür auto steht

Von auto zu Templates

- Vor C++20 sind auto-Funktionsparameter verboten

```
void print(auto x) {...} // Compiler error
```

Frage: Was meinen Sie, warum?
Antwort: Der Typ kann nicht vom Kontext abgeleitet werden

- Seit C++20 sind auto-Funktionsparameter erlaubt

```
void print(auto x) {...} // ok
```

Es ist natürlich immer noch nicht möglich, herauszufinden, wofür auto steht
Frage: Was bedeutet auto hier?

Von auto zu Templates

- Vor C++20 sind auto-Funktionsparameter verboten

```
void print(auto x) {...} // Compiler error
```

Frage: Was meinen Sie, warum?
Antwort: Der Typ kann nicht vom Kontext abgeleitet werden

- Seit C++20 sind auto-Funktionsparameter erlaubt

```
void print(auto x) {...} // ok
```

Es ist natürlich immer noch nicht möglich, herauszufinden, wofür auto steht
Frage: Was bedeutet auto hier?
Antwort: Abkürzung für ein Template-Parameter

```
template <typename T>
void Print(T x){ ... }
```

4. Wiederholung Theorie

Induktion: Was wird gebraucht?

- Beweise Aussagen, z.B. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Induktion: Was wird gebraucht?

- Beweise Aussagen, z.B. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Induktionsanfang:
  - Die gegebene Gleichung bzw. Ungleichung stimmt für einen oder mehrere Basisfälle.
  - z.B.: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$.

Induktion: Was wird gebraucht?

- Beweise Aussagen, z.B. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Induktionsanfang:
  - Die gegebene Gleichung bzw. Ungleichung stimmt für einen oder mehrere Basisfälle.
  - z.B.: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$.
- Induktionshypothese: Wir nehmen an, die Aussage stimmt für ein allgemeines n.

Induktion: Was wird gebraucht?

- Beweise Aussagen, z.B. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Induktionsanfang:
  - Die gegebene Gleichung bzw. Ungleichung stimmt für einen oder mehrere Basisfälle.
  - z.B.: $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$.
- Induktionshypothese: Wir nehmen an, die Aussage stimmt für ein allgemeines n.
- Induktionsschritt ($n \to n+1$):
  - Aus der Gültigkeit der Aussage für n (Induktionshypothese) folgt die Gültigkeit für $n+1$.
  - z.B.: $\sum_{i=1}^{n+1} i = n+1+\sum_{i=1}^{n} i = n+1+\frac{n(n+1)}{2} = \frac{(n+2)(n+1)}{2}$.

Induktion: Beispiel

- Zu zeigen: $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$.

Induktion: Beispiel

- Zu zeigen: $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$.
- Induktionsanfang:
  $n = 0$: $\sum_{i=0}^{0} r^i = 1 = \frac{1-r^1}{1-r}$.

Induktion: Beispiel

- Zu zeigen: $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$.
- Induktionsanfang:
  $n = 0$: $\sum_{i=0}^{0} r^i = 1 = \frac{1-r^1}{1-r}$.
- Induktionsschritt ($n \to n+1$):

$$\begin{aligned}\sum_{i=0}^{n+1} r^i &= r^{n+1} + \sum_{i=0}^{n} r^i \\ &= r^{n+1} + \frac{1-r^{n+1}}{1-r} = \frac{r^{n+1} - r^{n+2} + 1 - r^{n+1}}{1-r} = \frac{1-r^{n+2}}{1-r}.\end{aligned}$$

[Übrigens ..]

Das lässt sich auch einfach direkt zeigen

$$\frac{r^{n+1}-1}{r-1} \overset{!}{=} \sum_{i=0}^{n} r^i$$

$$\begin{aligned}(r-1)\cdot\sum_{i=0}^{n} r^i &= \sum_{i=0}^{n} r^{i+1} - \sum_{i=0}^{n} r^i \\ &= \sum_{i=1}^{n+1} r^i - \sum_{i=0}^{n} r^i = \sum_{i=0}^{n+1} r^i - 1 - \sum_{i=0}^{n} r^i \\ &= r^{n+1} - 1\end{aligned}$$

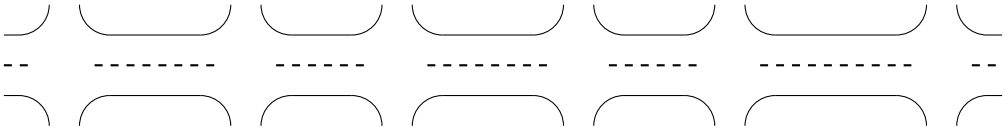
5. Use Case

5.1 Subarray Sum Problem

Naive Lösung, Präfixsummen, Binäre Suche, Sliding Window

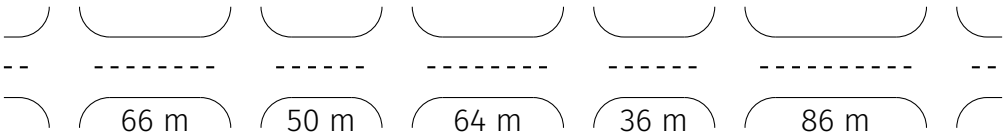
Strassenabschnitt einer bestimmten Länge

Strassenabschnitt einer bestimmten Länge



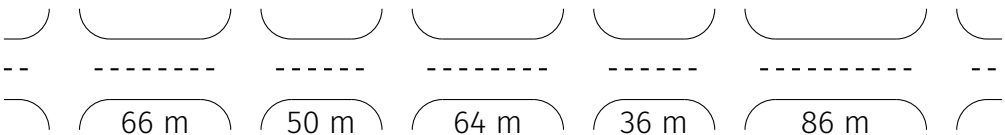
Strassenabschnitt einer bestimmten Länge

Gegeben: Distanzen zwischen Querstrassen auf Strasse



Strassenabschnitt einer bestimmten Länge

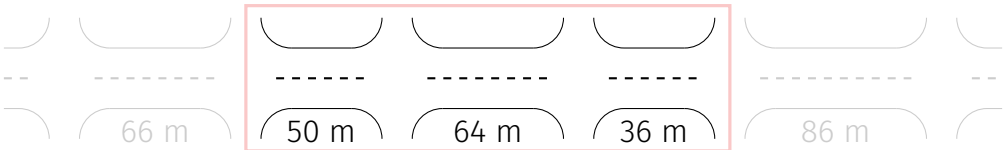
Gegeben: Distanzen zwischen Querstrassen auf Strasse



Gesucht: Strassenabschnitt der Länge 150 Meter zwischen Querstrassen

Strassenabschnitt einer bestimmten Länge

Gegeben: Distanzen zwischen Querstrassen auf Strasse



Gesucht: Strassenabschnitt der Länge 150 Meter zwischen Querstrassen

Subarray Sum Problem

Subarray Sum Problem

Gegeben: eine Sequenz $a[0], \ldots, a[n-1]$ nicht-negativer ganzer Zahlen

Gesucht: eine Subsequenz mit Summe k:

Paar (l, r) mit $0 \leq l \leq r \leq n-1$ so dass $\sum_{i=l}^{r} a[i] = k$

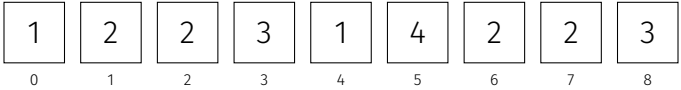
Subarray Sum Problem

Gegeben: eine Sequenz $a[0], \ldots, a[n-1]$ nicht-negativer ganzer Zahlen

Gesucht: eine Subsequenz mit Summe k:

Paar (l, r) mit $0 \leq l \leq r \leq n-1$ so dass $\sum_{i=l}^{r} a[i] = k$

Beispiel: $n = 9, k = 7$



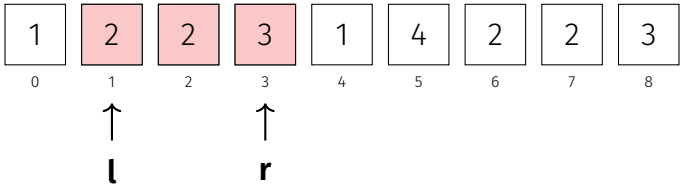
Subarray Sum Problem

Gegeben: eine Sequenz $a[0], \ldots, a[n-1]$ nicht-negativer ganzer Zahlen

Gesucht: eine Subsequenz mit Summe k:

Paar (l, r) mit $0 \leq l \leq r \leq n-1$ so dass $\sum_{i=l}^{r} a[i] = k$

Beispiel: $n = 9, k = 7$ **Lösung:** $l = 1, r = 3$.



Strategien?

Gegeben: eine Sequenz $a[0], \ldots, a[n-1]$ nicht-negativer ganzer Zahlen

Gesucht: eine Subsequenz mit Summe k:

Paar (l, r) mit $0 \leq l \leq r \leq n-1$ so dass $\sum_{i=l}^{r} a[i] = k$

Strategien

$\Theta(n^3)$	Drei Schleifen
$\Theta(n^2)$	?
$\Theta(n \log n)$	?
$\Theta(n)$	?

Strategien?

Gegeben: eine Sequenz $a[0], \ldots, a[n-1]$ nicht-negativer ganzer Zahlen

Gesucht: eine Subsequenz mit Summe k:

Paar (l, r) mit $0 \leq l \leq r \leq n-1$ so dass $\sum_{i=l}^{r} a[i] = k$

Strategien

$\Theta(n^3)$	Drei Schleifen
$\Theta(n^2)$	Präfixsummen
$\Theta(n \log n)$	?
$\Theta(n)$	?

Strategien?

Gegeben: eine Sequenz $a[0], \ldots, a[n-1]$ nicht-negativer ganzer Zahlen

Gesucht: eine Subsequenz mit Summe k:

Paar (l, r) mit $0 \leq l \leq r \leq n-1$ so dass $\sum_{i=l}^{r} a[i] = k$

Strategien

$\Theta(n^3)$	Drei Schleifen
$\Theta(n^2)$	Präfixsummen
$\Theta(n \log n)$	Binäre Suche
$\Theta(n)$	?

Strategien?

Gegeben: eine Sequenz $a[0], \ldots, a[n-1]$ nicht-negativer ganzer Zahlen

Gesucht: eine Subsequenz mit Summe k:

Paar (l, r) mit $0 \leq l \leq r \leq n-1$ so dass $\sum_{i=l}^{r} a[i] = k$

Strategien

$\Theta(n^3)$	Drei Schleifen
$\Theta(n^2)$	Präfixsummen
$\Theta(n \log n)$	Binäre Suche
$\Theta(n)$	Sliding Window

Subarray Sum Problem: Sliding Window

Sliding Window Idee

Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten

Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:

Subarray Sum Problem: Sliding Window

Sliding Window Idee

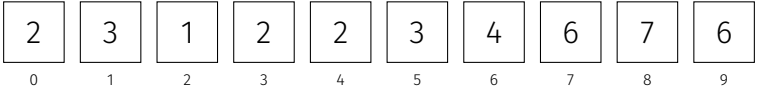
- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$

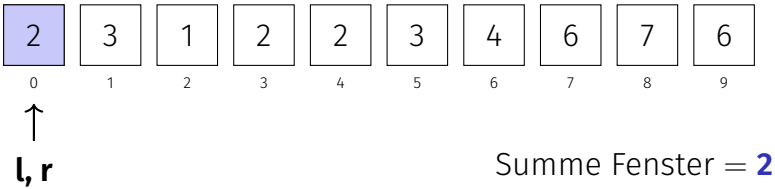


Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$

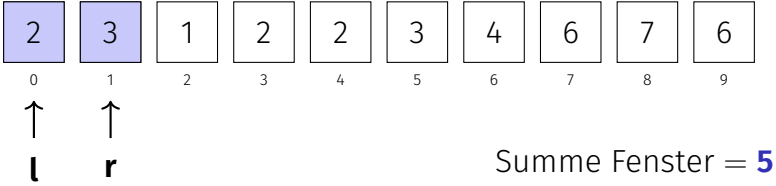


Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$



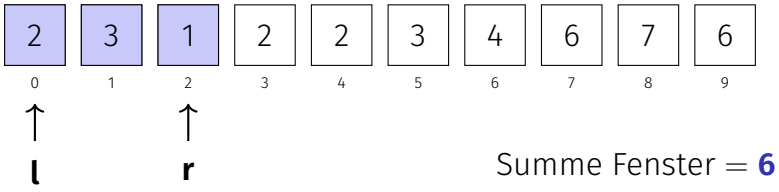
Summe Fenster $= \mathbf{5}$

Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$

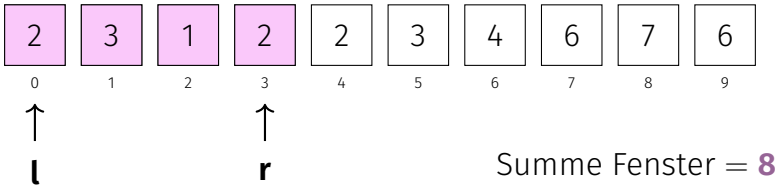


Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$

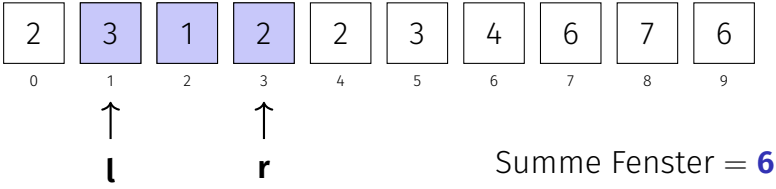


Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$

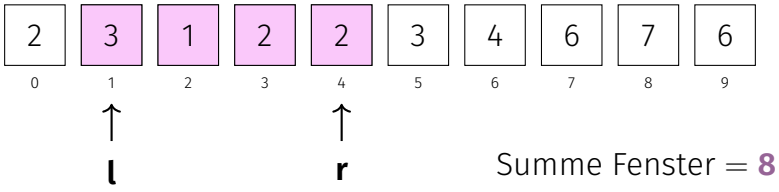


Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$

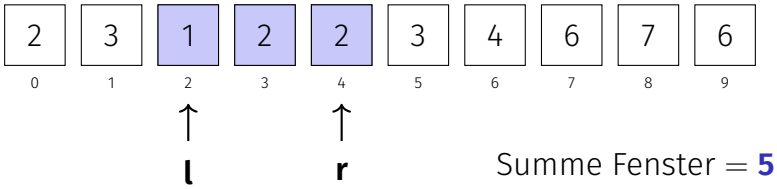


Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$

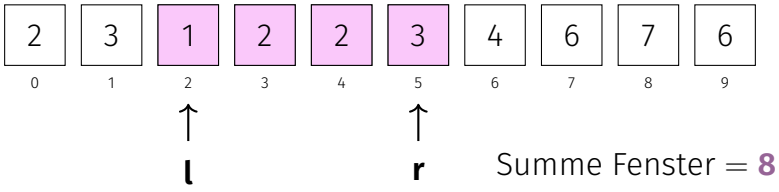


Subarray Sum Problem: Sliding Window

Sliding Window Idee

- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$

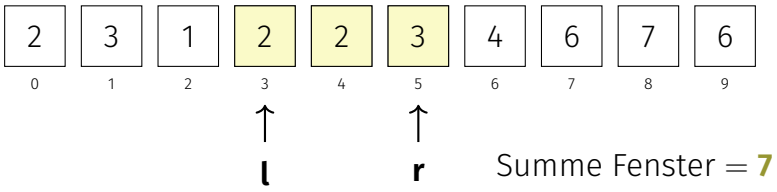


Subarray Sum Problem: Sliding Window

Sliding Window Idee

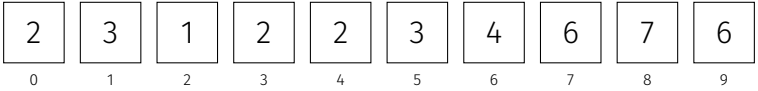
- mit linkem und rechtem Zeiger bei 0 starten
- bis zum Ende der Sequenz wiederholen:
  - Fenster **zu klein** (Summe $< k$) $\Rightarrow$ rechten Zeiger erhöhen
  - Fenster **zu gross** (Summe $> k$) $\Rightarrow$ linken Zeiger erhöhen
  - Fenster **wie gewünscht** (Summe $= k$) $\Rightarrow$ fertig!

Beispiel: $k = 7$



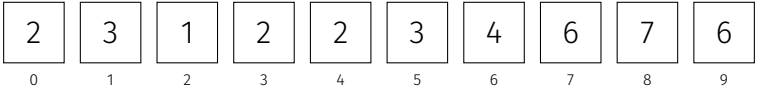
Subarray Sum Problem: Sliding Window Analyse

Subarray Sum Problem: Sliding Window Analyse



Subarray Sum Problem: Sliding Window Analyse

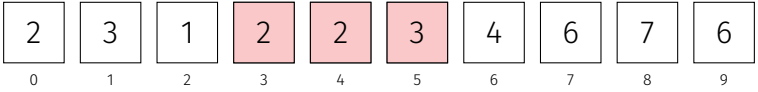
- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten



Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

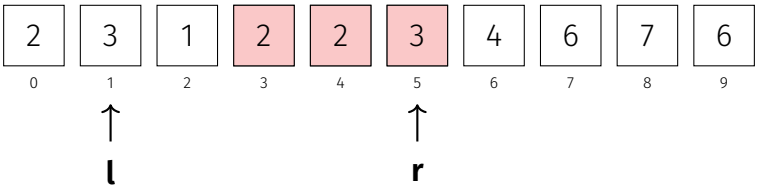


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand

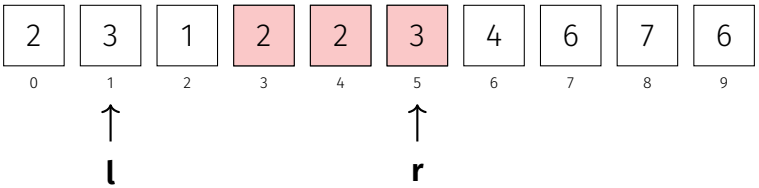


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross

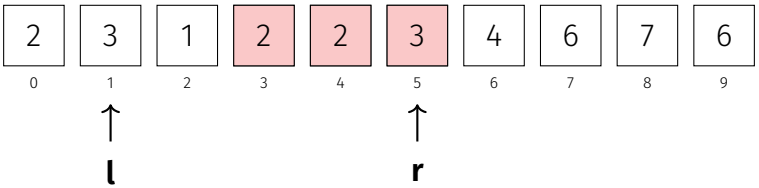


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross $\Rightarrow$ l wird erhöht bis zum linken Rand

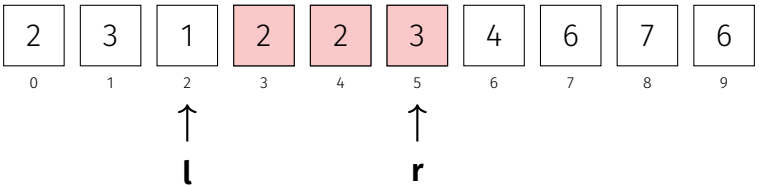


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross $\Rightarrow$ l wird erhöht bis zum linken Rand

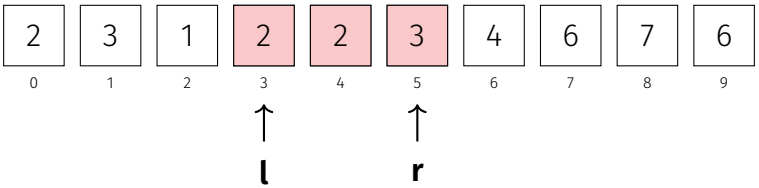


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross $\Rightarrow$ l wird erhöht bis zum linken Rand

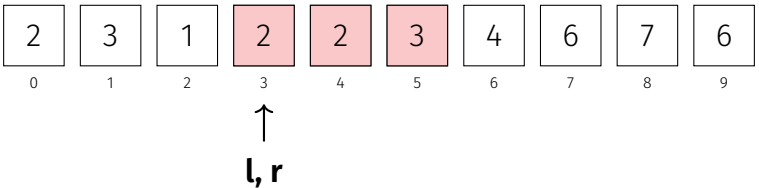


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross $\Rightarrow$ l wird erhöht bis zum linken Rand
- falls l den linken Rand des Zielfensters erreicht bevor r den rechten Rand

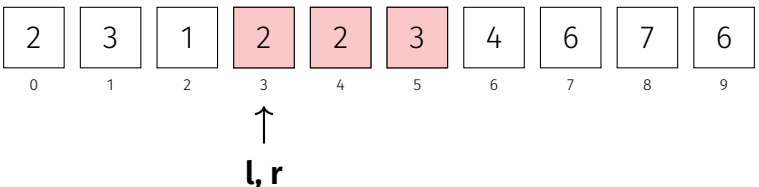


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross $\Rightarrow$ l wird erhöht bis zum linken Rand
- falls l den linken Rand des Zielfensters erreicht bevor r den rechten Rand
  $\Rightarrow$ Summe zu klein

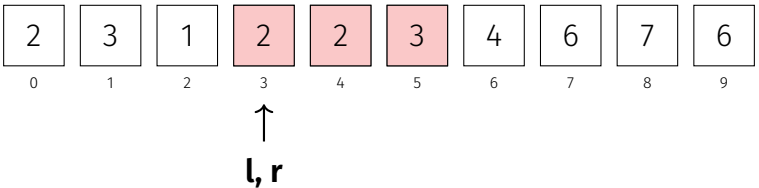


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross $\Rightarrow$ l wird erhöht bis zum linken Rand
- falls l den linken Rand des Zielfensters erreicht bevor r den rechten Rand
  $\Rightarrow$ Summe zu klein $\Rightarrow$ r wird erhöht bis zum rechten Rand

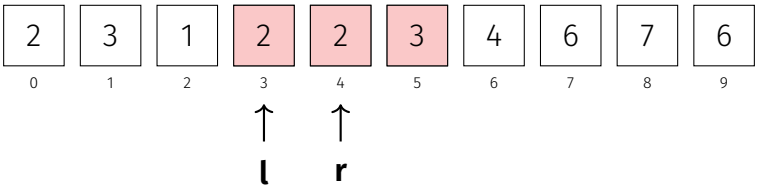


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross $\Rightarrow$ l wird erhöht bis zum linken Rand
- falls l den linken Rand des Zielfensters erreicht bevor r den rechten Rand
  $\Rightarrow$ Summe zu klein $\Rightarrow$ r wird erhöht bis zum rechten Rand

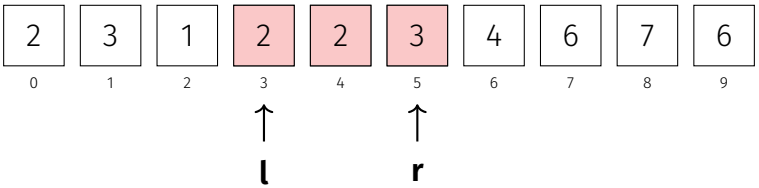


Subarray Sum Problem: Sliding Window Analyse

- in jedem Schritt: l oder r erhöht
  $\Rightarrow$ Algorithmus terminiert nach maximal $2n$ Schritten

Zielfenster: lexikographisch kleinstes (linkestes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
  $\Rightarrow$ Summe zu gross $\Rightarrow$ l wird erhöht bis zum linken Rand
- falls l den linken Rand des Zielfensters erreicht bevor r den rechten Rand
  $\Rightarrow$ Summe zu klein $\Rightarrow$ r wird erhöht bis zum rechten Rand



Analyse

Wir betrachten das lexikographisch kleinste (linkeste) Fenster mit Summe k, genannt *Zielfenster*

- In jedem Schritt des Algorithmus wird l oder r erhöht. Der Algorithmus terminiert nach maximal $2n$ Schritten.
- Angenommen r erreicht den rechten Rand des Zielfensters bevor l den linken Rand erreicht hat, dann wird i weiterhin erhöht bis es den linken Rand des Fensters erreicht.
- Angenommen l erreicht den linken Rand des Zielfensters bevor r den rechten Rand erreicht hat, dann wird r weiterhin erhöht bis es den rechten Rand des Fensters erreicht.

Übung: Fenster mit Summe am nächsten an k

Fragen oder Anregungen?