



D&A - Übungsstunde 2

Diese Folien basieren auf denjenigen der Vorlesung, wurden aber durch den Assistenten Adel Gavranović adaptiert und erweitert

Übersicht

Heutiges Programm

Follow-up

Kleinigkeiten zu `[code]expert`

Feedback letzte Übung

C++ Container Bibliothek

Templates Wiederholung

Wiederholung Theorie: Induktion

Use Case: Subarray Sum Problem

Outro



`n.ethz.ch/~agavranovic`

▶ [Link zum Material für die Übungsstunden](#)

▶ [Webseite des Assistenten](#)

▶ [Mail an Assistenten](#)

Follow-up aus vorheriger Übungsstunde

Fehler meinerseits:

- Die annotierten Folien aus der ersten Übungsstunde wurden nicht hochgeladen, weil sie ziemlich schlecht annotiert waren

Frage aus Letzter Übungsstunde

Frage

- Sind alle Funktionen in $\Theta(g)$ (für eine gegebene Funktion g) einfach nur ein $c \cdot g$ für $c \in \mathbb{R}$?

Antwort

- Nein, siehe [code]expert -Aufgabe "The Set Theta(g)"
- Beispiel
 - $f(n) = 84n \cdot \log(n)$
 - $g(n) = 2n \cdot \log(n) + n$, also $\Theta(n \cdot \log(n))$
 - $f \in \Theta(g)$

Aber stimmt $f \in \Theta(g)$ wirklich?

Sicher, dass f in $\Theta(g)$ ist?

- Nein, deshalb überprüfen wir das:

$$\begin{aligned} & \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} && | \text{Funktionen einsetzen} \\ &= \lim_{n \rightarrow \infty} \frac{84n \cdot \log(n)}{2n \cdot \log(n) + n} && | \cdot \frac{\frac{1}{n}}{\frac{1}{n}} \\ &= \lim_{n \rightarrow \infty} \frac{84 \cdot \log(n)}{2 \cdot \log(n) + 1} && | \text{Konstante rausziehen} \\ &= \frac{84}{2} \cdot \lim_{n \rightarrow \infty} \frac{\log(n)}{\log(n) + 0.5} && | n \rightarrow \infty \\ &= \frac{84}{2} \cdot 1 = 42 \in \mathbb{R} \implies f \in \Theta(g) \end{aligned}$$

Übungsbedarf?

Die letzten paar Seiten der Slides von letzten Woche haben viele Übungen wie diese

y -Achsenabschnitt

Frage

- *Was hat es mit dem y -Achsenabschnitt auf sich?*

Antwort

- Nichts, vergesst ihn einfach
- (Ausser jemand möchte mir nach der Lektion zeigen wieso er doch wichtig ist)

nach n_0 und c lösen

Frage

- *Wie berechne ich n_0 und c für gegebene $f(n)$ und $\mathcal{O}(g)$?*

Antwort

- Mit Mittel der Analysis ;)
- Falls euch ein solides Rezept einfällt, lasst es mich wissen
- Vergesst nicht, dass nur die Existenz von n_0 und c erforderlich ist

War da nicht noch was wegen den Laufzeiten?

Naja...

- Es gab scheinbar Fehler bei den Lösungen für die Laufzeiten der Programme auf den Folien der letzten Übungsstunde
- Ich bin noch dran, sie zu melden/bestätigen
- Wir werden sie uns baldmöglichst nochmal anschauen

Zusammenfassung für die Prüfung

Aus dem Vorlesungsverzeichnis:

- *Sie dürfen maximal 4 A4-Blätter mit in die Prüfung nehmen*
- *Inhaltliche und formale Anforderungen (Text, Bilder, ein-/doppelseitig, Ränder, Schriftgrösse, etc.) bestehen nicht*
- Das sehr gute Voraussetzungen für gute Zusammenfassungen!

Wie sah meine Zusammenfassung aus?

Kommentare zu [code]expert

Konkretes Feedback

- Deadline war auf heute
- Dementsprechend noch kein konkretes Feedback dazu bereit

Welcher log?

Es gilt:

$$\log_a(n) \in \Theta(\log_b(n))$$

(siehe Musterlösung zu [code]expert -Aufgabe "Some Proofs")

Das heisst für uns: alle logs sind quasi gleichwertig

Fragen zu `[code]expert` eurerseits?

Allgemein gilt:

- Schreibt mir eine Mail
- Schreibt sie direkt in den (vielleicht nicht funktionierenden) Code
 - am besten gleich dort, wo man etwas nicht verstanden hat
- oder ladet eine PDF/txt/md-Datei ins `[code]expert` hoch

Landau-Notation

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

Landau-Notation

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

(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

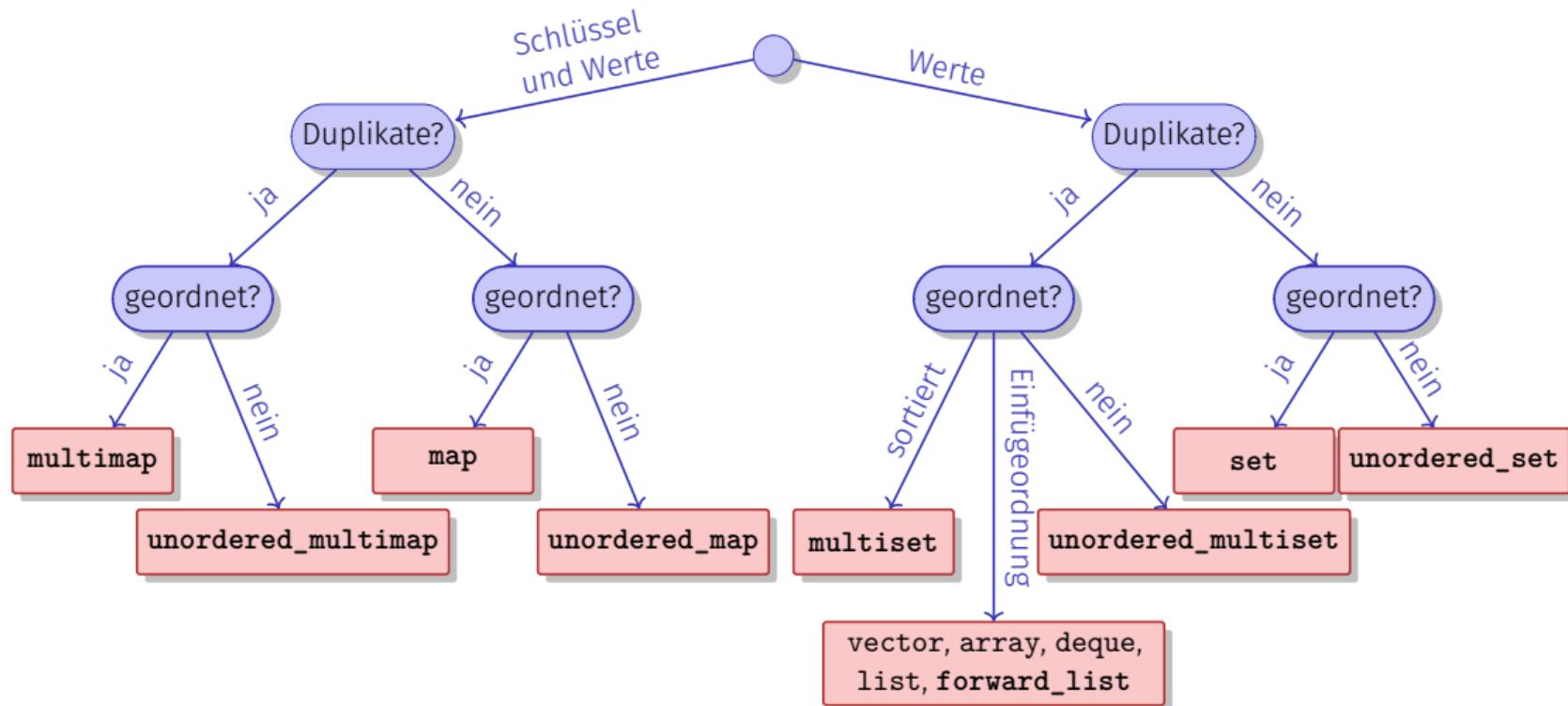
Funktionen sortieren: wenn Funktion f links der Funktion g steht, dann $f \in \mathcal{O}(g)$.

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 .

Fragen/Unklarheiten?

4. C++ Container Bibliothek

C++ Container



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 Zu- griff	Wahlfreier Zu- griff	Wahlfreier Zu- griff		
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

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 Schlüsselanzahl (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
```

Fragen/Unklarheiten?

5. 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
```

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)
    }
    ...
};
```

Funktientemplate: 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

```
Node<int> n1 = ...;  
Node<std::string> n2 = ...;  
Node<Student> n3 = ...;
```

n1

```
class Node_int {  
    int key;  
    ...  
    bool contains(int k) {...}  
    int max() {...}  
};
```

n2

```
class Node_string {  
    std::string key;  
    ...  
};
```

n3

```
class Node_Student {  
    Student key;  
    ...  
};
```

Semantik (Code-Generation)

Für jede Instanz eines templates generiert der Compiler eine entsprechende Klasse (oder Funktion) → 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
abs(8); // OK
```

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

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

5.1 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 std::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 instanziiert 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?

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
```

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){ ... }
```

Fragen/Unklarheiten?

6. Wiederholung Theorie: Induktion

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 ($n \rightarrow n + 1$):
 - 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: $\sum_{i=0}^n r^i = \frac{1-r^{n+1}}{1-r}$.

■ Induktionsanfang:

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

■ Induktionsschritt ($n \rightarrow n + 1$):

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

[Übrigens ..]

Das lässt sich auch einfach direkt zeigen

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

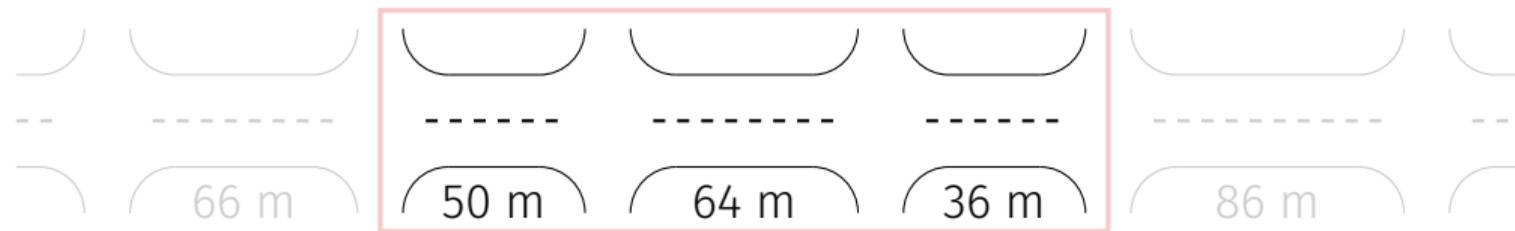
Fragen/Unklarheiten?

7. Use Case: Subarray Sum Problem

Naive Lösung, Präfixsummen, Binäre Suche, Sliding Window

Strassenabschnitt einer bestimmten Länge

Gegeben: Distanzen zwischen Querstrassen auf Strasse



Gesucht: Strassenabschnitt der Länge 150 Meter zwischen Querstrassen

Strategien?

Gegeben: eine Sequenz $a[0], \dots, a[n - 1]$ 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

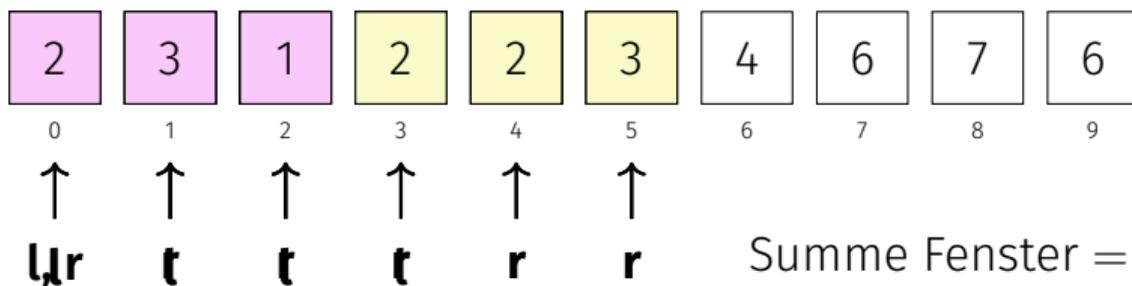
$\Theta(n^3)$	Drei Schleifen
$\Theta(n^2)$	Präfixsummen
$\Theta(n \log n)$	Binäre Suche
$\Theta(n)$	Sliding Window

Subarray Sum Problem: Sliding Window

Sliding Window Idee

- 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: $k = 7$

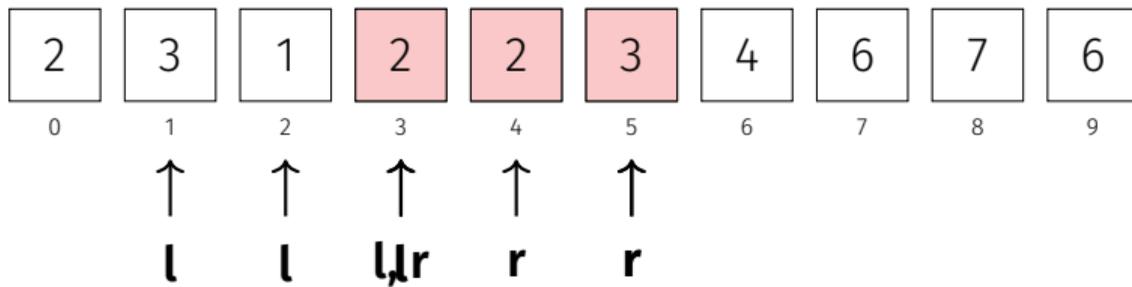


Subarray Sum Problem: Sliding Window Analyse

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

Zielfenster: lexikographisch kleinstes (linkstes) Fenster mit Summe k

- falls r den rechten Rand erreicht bevor l den linken Rand
⇒ Summe zu gross ⇒ l wird erhöht bis zum linken Rand
- falls l den linken Rand des Zielfensters erreicht bevor r den rechten Rand
⇒ Summe zu klein ⇒ 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/Unklarheiten?

Tipps für [code]expert

Markdown und \LaTeX

- Übt \LaTeX , wenn ihr Zeit habt ([▶ detexify](#))
- Markdown kann sehr nützlich sein
- Falls wenig Zeit: ein PDF tuts auch

Allgemeine Fragen?

Bis zum nächsten Mal

Schönes Wochenende!

(es gibt heute vegan (planted) Poutine in der Polymensa)