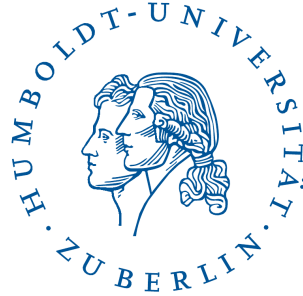


Übung Algorithmen und Datenstrukturen



Sommersemester 2017

Patrick Schäfer, Humboldt-Universität zu Berlin

Agenda: Suchen und Amortisierte Analyse

- **Heute:**

- Suchen / Schreibtischtest
- Amortisierte Analyse

- Nächste Woche: Vorrechnen (first-come first-served)

- Gruppe 5 13-15 Uhr <https://dudle.inf.tu-dresden.de/AlgoDatGr5U3/>
- Gruppe 6 15-17 Uhr <https://dudle.inf.tu-dresden.de/AlgoDatGr6U3/>

Übung: <https://hu.berlin/algodat17>

Vorlesung: https://hu.berlin/vl_algodat17

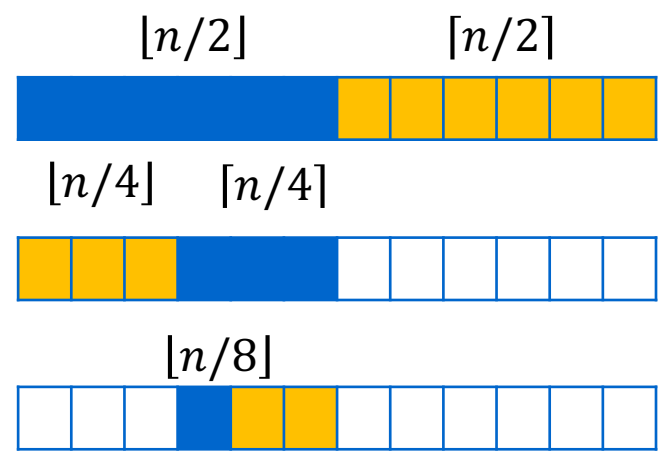
Organisatorisches

- Abholung alter Übungsaufgaben:
 - Montag, der 12.06. 14:45 - 15:15 RUD 25, 3.321
- Tutorium (amortisierte Analyse, etc.)
 - Zusätzlich zu den Übungsterminen gibt es noch ein Tutorium
 - Montag 17:00 - 19:00 s.t. RUD 26, 1'303
 - Freitag 11:00 - 13:00 c.t. RUD 26, 1'303, Stefanie Lowski
 - In dieser Woche kann leider am Freitag (9.6.) kein Tutorium stattfinden
 - Ersatztermin: heute, am Mittwoch (7.6.), von 17:00-18:30, RUD 26, 1'303.

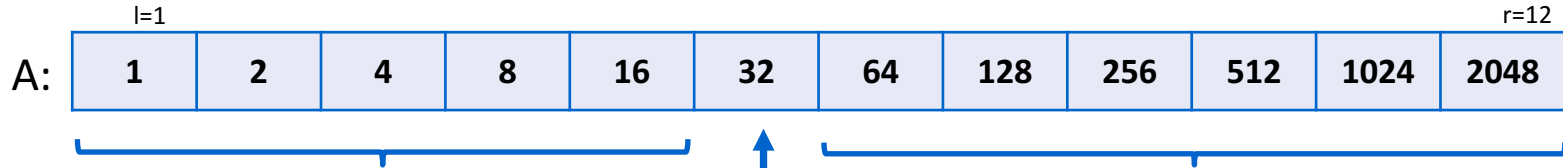
Suche in sortierten Arrays

- Suchverfahren aus der Vorlesung:
 - Binäre Suche (Aufgabe 2)**

$$m = \left\lfloor \frac{(r + l)}{2} \right\rfloor = l + \left\lfloor \frac{(r - l)}{2} \right\rfloor$$



$c=8$



$$m = \left\lfloor \frac{(12 + 1)}{2} \right\rfloor = 6$$

Suche: Schreibtischttest

Führen Sie einen Schreibtischttest für die *binäre Suche* durch, bei dem das folgende Array A nach dem Wert $c = 68$ durchsucht wird.

Geben Sie dazu an, mit welchen Werten die Variablen l , r und m nach jedem Aufruf von Zeile 4 belegt sind.

$A = [5, 12, 15, 17, 22, 29, 45, 47, 60, 61, 68, 74, 77]$

l	r	m
1	13	$14/2=7$
...

Algorithmus BinarySearch(A, c)

Input: Sortiertes Array A und Integer c

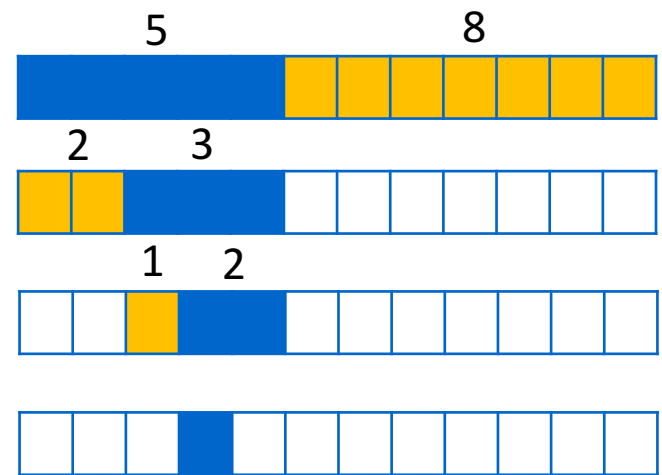
Output: TRUE, falls das Element c in A ist.

```
1:  $l := 1$ ;  
2:  $r := |A|$ ;  
3: while  $l \leq r$  do  
4:    $m := (l + r) \text{ div } 2$ ;  
5:   if  $c < A[m]$  then  
6:      $r := m - 1$ ;  
7:   else if  $c > A[m]$  then  
8:      $l := m + 1$ ;  
9:   else  
10:    return true;  
11:  end if  
12: end while  
13: return false;
```

Suche in sortierten Arrays

- Suchverfahren aus der Vorlesung:
 - Fibonacci-Suche (Aufgabe 1)**

$$F_1 = 1, F_2 = 2, F_k = F_{k-2} + F_{k-1}$$



$c=8$

	$l=1$												
A:	1	2	4	8	16	32	64	128	256	512	1024	2048	$r=12$

$$fib2 \sim \frac{1}{3}(r - l)$$

$$fib1 \sim \frac{2}{3}(r - l)$$

$$m_{fib} = \min(fib2, n)$$

$$fib=13, fib1=8, fib2=5$$

Suche: Schreibtischttest

Führen Sie einen Schreibtischttest für die *Fibonacci-Suche* durch, bei dem das folgende Array A nach dem Wert $c = 34$ durchsucht wird. Geben Sie die aktuellen Belegungen der Variablen $fib2$, $fib3$, und m vor jedem Aufruf von Zeile 8 im Pseudocode von Folie 13 an.

$$A = [5, 6, 9, 10, 12, 13, 34, 39, 43, 52, 63, 76]$$

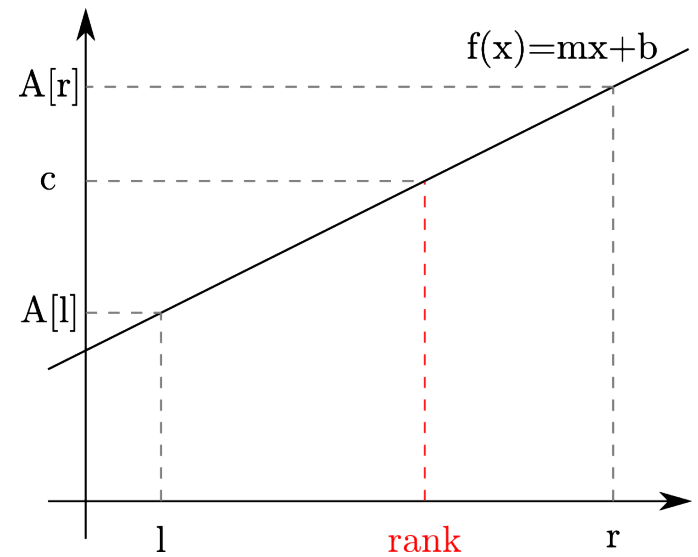
fib2	fib3	m
fib(5)=5	fib(4)=3	5
...

```
1. A: sorted_int_array;
2. c: int;
3. compute i; #fib(i) smallest ...
4. fib3 := fib(i-3);
5. fib2 := fib(i-2);
6. m := fib2;
7. repeat
8.   if c>A[m] then
9.     if fib3=0 then return false
10.    else
11.      m := m+fib3;
12.      tmp := fib3;
13.      fib3 := fib2-fib3;
14.      fib2 := tmp;
15.    end if;
16.  else if c<A[m]
17.    if fib2=1 then return false
18.    else
19.      m := m-fib3;
20.      fib2 := fib2 - fib3;
21.      fib3 := fib3 - fib2;
22.    end if;
23.  else return true;
24. until true;
```

Suche in sortierten Arrays

- Suchverfahren aus der Vorlesung:
 - Interpolations-Suche (Aufgabe 2)**

$$\text{rank} = l + \frac{(r - l)(c - A[l])}{A[r] - A[l]}$$



$c=8$

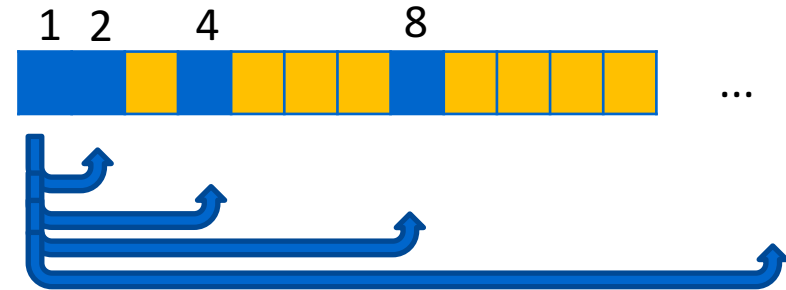
	$l=1$										$r=12$	
A:	1	2	4	8	16	32	64	128	256	512	1024	2048

$$\text{rank} = 1 + \left\lfloor \frac{(12-1)(8-1)}{2048-1} \right\rfloor = 1$$

Suche in sortierten Arrays

- Suchverfahren aus der Vorlesung:
 - **Exponentielle Suche (Aufgabe 2):**
 - Suche zunächst rechten Rand.
 - Suche im Suchbereich mit binärer Suche

$$A[2^i] \leq c < \min(A[2^{i+1}], n)$$



$c=16$

	$l=1$										$r=12$	
A:	1	2	4	8	16	32	64	128	256	512	1024	2048



$$A[2^2] \leq c < A[2^3]$$

Aufgabe 2: SortedSearch.java

- Unsere Vorlage verwendet ein **Comparator-Objekt**, das mitzählt, wie oft verglichen wird.
- Ein Comparator in JAVA liefert
 - >0 , falls $\text{element} > \text{key}$
 - $=0$, falls $\text{element} == \text{key}$
 - <0 , falls $\text{element} < \text{key}$

```
public static class LinearSearch extends Search {  
  
    @Override  
    public boolean search(Long[] sortedList, Long key) {  
        for (Long element : sortedList) {  
            // compare returns a negative integer, zero, or  
            // a positive integer if the first argument is  
            // less than, equal to, or greater than the second.  
            int comparison = this.comparator.compare(element, key);  
            if (comparison == 0) {  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

Agenda: Suchen und Amortisierte Analyse

- Suchen / Schreibtischtest
- **Amortisierte Analyse**
 - **Binärzähler**
 - Account-Methode
 - Potentialmethode

Binärzähler

- Geg.: k -Bit Binärzähler
- Entspricht Binärzahl $b_{k-1} \dots b_1 b_0$ bzw. Dezimalzahl $\sum_i b_i 2^i$
- **Operation**: Zahl inkrementieren (um 1 erhöhen)
- **Kosten**: Anzahl der Bitänderungen (jedes Bit kostet 1)

n	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	0
1	0	0	0	0	1

↪ 1 Bitwechsel


n	b_4	b_3	b_2	b_1	b_0
23	1	0	1	1	1
24	1	1	0	0	0

↪ 4 Bitwechsel

Binärzähler – Kostenabschätzung

- **Gesucht:** Kosten für n Inkrement-Operationen, wenn Zähler bei 0 beginnt.
 - **Best Case:** Eine Bitänderung $\Omega(n)$
 - **Worst Case:** Alle k Bits werden verändert: $O(nk)$
- **Problem:** sehr **pessimistische Abschätzung**, da Worst Case eher selten vorkommt

n	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
127	0	1	1	1	1	1	1	1
128	1	0	0	0	0	0	0	0

 k Bitwechsel

n	b_3	b_2	b_1	b_0	#BW
0	0	0	0	0	-
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	1
4	0	1	0	0	3
5	0	1	0	1	1
6	0	1	1	0	2
7	0	1	1	1	1
8	1	0	0	0	4
9	1	0	0	1	1
10	1	0	1	0	2

⇒ Häufig geringe Kosten

Summe der Kosten

$T(n)$	$\Sigma\#BW$	$n \cdot k$
n=1	1	4
2	3	8
3	4	12
4	7	16
5	8	20
6	10	24
7	11	28
8	15	32
9	16	36
10	18	40

⇒ Abschätzung zu pessimistisch

Amortisierte Analyse

- Kosten werden über eine Sequenz von Operationen gemittelt.
- Amortisierte Analyse beschreibt **mittlere Kosten einer Operation** im schlechtesten Fall (**Worst-Case**).
- Grundidee von **Account-** und **Potentialmethode**: Anfänglich günstige Operation teurer bewerten, um spätere (teure) Operationen auszugleichen.
 - Überschuss wird als Kredit/Potential gespeichert.
 - Kredit/Potential wird verwendet, um für teurere Operation zu zahlen.
 - Verfahren unterstützen mehr als einen Operationstypen.

Account-Methode (Bankkonto~, Guthaben~)

- Idee: *Operationen* werden Kosten zugewiesen, wobei für einige mehr und für andere weniger als die *tatsächlichen Kosten* berechnet wird. Diese zugewiesenen Kosten nennen wir *amortisierte Kosten*.
- Übersteigen die amortisierten Kosten die tatsächlichen Kosten, so wird die Differenz als *Kredit* in der Datenstruktur gespeichert.
- Der *Kredit* ist definiert als die Differenz zwischen den *amortisierten Kosten* mit \hat{c}_i und den *tatsächlichen Kosten* c_i der i -ten Operation:

$$\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i \geq 0 \quad (\text{nichtnegativ})$$

- Der Kredit muss zu allen Zeiten *nichtnegativ* sein, damit die amortisierten Kosten eine obere Schranke der Gesamtkosten bilden.
- Dieser Kredit kann später verwendet werden, um Operation zu bezahlen, deren tatsächliche Kosten höher als die amortisierten Kosten sind.

Binärzähler (Account-Methode)

- **Tatsächliche Kosten:** Wir verwenden 1\$, um das Kippen eines Bits zu bezahlen.
- Idee: Der gespeicherte **Kredit** soll der Anzahl 1en im Zähler entsprechen.
- Wir berechnen:
 - **2\$**, um irgendein Bit von **0 auf 1** zu kippen. 1\$ für das Setzen des Bits und 1\$ für das (spätere) Rücksetzen auf 0.
 - **0\$**, um irgendein Bit von **1 auf 0** zu setzen. Diese Kosten wurden bereits mit dem Setzen verrechnet.
- **Amortisierte Kosten:** Es kippt (maximal) eine 0 zu einer 1: entweder direkt an Stelle b_0 oder als Übertrag. Die amor. Kosten sind somit: $\hat{c}_i \leq 2$.
- **Nichtnegativ:** Zu jedem Zeitpunkt hat jede 1 im Zähler 1\$ Kredit, um das Rücksetzen zu bezahlen. Somit befinden sich immer ausreichend \$ im Kredit, um das Rücksetzen eines Bits zu bezahlen.

⇒ Die amortisierten Kosten \hat{c}_i bilden eine obere Schranke der tatsächlichen Kosten c_i :

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i \leq \sum_{i=1}^n 2 = 2n$$

⇒ Und somit ist die **Worst-Case Komplexität** $O(n)$

n	b_3	b_2	b_1	b_0	c_i	\hat{c}_i	Kredit
0	0	0	0	0	-	-	0
1	0	0	0	1	1	2	1
2	0	0	1	0	2	2	1
3	0	0	1	1	1	2	2
4	0	1	0	0	3	2	1
5	0	1	0	1	1	2	2
6	0	1	1	0	2	2	2
7	0	1	1	1	1	2	3
8	1	0	0	0	4	2	1
9	1	0	0	1	1	2	2
10	1	0	1	0	2	2	2
				Σ	18	20	

Potentialmethode

- Die vorausbezahlten Kosten werden *in der Datenstruktur D* als *Potential* gespeichert, das freigegeben werden kann für zukünftige Operationen.
- Die *Potentialfunktion* Φ bildet die Datenstruktur D_i auf eine reelle Zahl ab, welche dieses Potential von D nach der *i -ten Operation* repräsentiert.

- Die *amortisierten Kosten* \hat{c}_i der i -ten Operation sind somit definiert als:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- D.h., die amortisierten Kosten von *n Operationen* sind:

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)$$

- Diese bilden eine obere Schranke der tatsächlichen Kosten, wenn gilt:

$$\Phi(D_i) - \Phi(D_0) \geq 0, \text{ für alle } i \quad (\textit{nichtnegativ})$$

Potentialmethode - Herangehensweise

- **Ziel:** Finde eine Potentialfunktion Φ , so dass
 1. $\Phi(D)$ von einer Eigenschaft von D abhängt,
 2. $\Phi(D_i) \geq \Phi(D_0)$ (**nichtnegativ**),
 3. \hat{c}_i lässt sich für alle i berechnen.
- Dann sind die amortisierten Gesamtkosten **obere Schranke** für die tatsächlichen Gesamtkosten
- Herangehensweise: Ermitteln einer Potentialfunktion, die hinreichend (aber nicht unnötig viel) **Potential** für spätere teure Operationen **sammelt**

Binärzähler (Potentialmethode)

- Idee: Das Potential des Zählers entspricht der im Zähler gespeicherten Einsen nach der i -ten Operation:
 $\Phi(D_i) = \text{Anzahl 1en in } D_i \text{ nach der } i\text{-ten Operation}$
- Das Potential $\Phi(D_i)$ hängt von D ab und ist anfangs $\Phi(D_0) = 0$
- Das Potential ist *nichtnegativ*, denn es gilt für alle i :
 $\Phi(D_i) - \Phi(D_0) = \Phi(D_i) \geq 0$
- Wir berechnen nun die *amortisierten Kosten* \hat{c}_i :
 - $\Phi(D_{i-1})$: Angenommen D_{i-1} endet auf l Einsen und enthält insgesamt $l + z$ Einsen
 - $\Phi(D_i)$: Nun kippen in D_i l Einsen auf 0. Zusätzlich springt (maximal) eine 0 auf eine 1 (Übertrag).
 Beispiel: $1101 + 1 = 1110$
- Es gilt:
 $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (l + 1) + (z + 1) - (z + l) = 2$
- Es gilt weiter: $\sum c_i = \sum \hat{c}_i - \Phi(D_n) + \Phi(D_0) \leq 2n$ und somit ist die *Worst-Case Komplexität* $O(n)$

n	b_3	b_2	b_1	b_0	c_i	$\Phi(D_i)$	\hat{c}_i
0	0	0	0	0	-	0	-
1	0	0	0	1	1	1	1+1-0=2
2	0	0	1	0	2	1	2+1-1=2
3	0	0	1	1	1	2	1+2-1=2
4	0	1	0	0	3	1	3+1-2=2
5	0	1	0	1	1	2	1+2-1=2
6	0	1	1	0	2	2	2+2-2=2
7	0	1	1	1	1	3	1+3-2=2
8	1	0	0	0	4	1	4+1-3=2
9	1	0	0	1	1	2	1+2-1=2
10	1	0	1	0	2	2	2+2-2=2
				Σ	18	20	