



# Informatik I

Prof. Dr. Christian Pape

## Kapitel 11

### Suchalgorithmen, Aufwandsabschätzungen



# Inhalt

- Suchalgorithmen
  - Sequentielle Suche
  - Zeitaufwand, Aufwandsabschätzung
  - Binärsuche



# Suchproblem / Sequentielle Suche

- Gegeben

Folge von aufsteigend sortierter Zahlen und eine weitere Zahl  $z$

- Gesucht

Ist  $z$  in der Folge der Zahlen enthalten?

---

Folge von Zahlen sei in einem Feld  $a$  gegeben

4	5	6	8	11	12	14	21	25	27	32	42
---	---	---	---	----	----	----	----	----	----	----	----

$z = 11$



# Suchproblem / Sequentielle Suche

- Sequentielle Suche
  - Gehe von links nach rechts durch das Feld
  - Wenn  $z$  gefunden ist, dann gib true zurück
- Wenn alle Elemente durchwandert wurden
  - Dann gibt false zurück

Folge von Zahlen sei in einem Feld  $a$  gegeben

4	5	6	8	11	12	14	21	25	27	32	42
---	---	---	---	----	----	----	----	----	----	----	----

↑  
 $i$

$z = 11$

# Suchproblem / Sequentielle Suche

```
public class Zahlenfolge {

    private int a[];

    public Zahlenfolge(int a[]) {
        this.a = a;
    }
}
```

## Zahlenfolge

-a[] : int

+Zahlenfolge(a [] : int)

+suchen(z : int) : boolean

```
public boolean suchen(int z) {
    for (int zahl : a) {
        if (zahl == z) {
            return true;
        }
    }

    return false;
}
```

A

Wie gut (schnell) ist diese Suche?

# Suchproblem / Sequentielle Suche

- Sequentielle Suche
  - Gehe von links nach rechts durch das Feld
  - Wenn  $z$  gefunden ist, dann gib true zurück
  - **Wenn nur noch Zahlen  $> z$  vorhanden sind, dann gibt false zurück**
- Wenn alle Elemente durchwandert wurden
  - Dann gibt false zurück

Suche kann bei 21 abgebrochen werden

4	5	6	8	11	12	14	21	25	27	32	42
---	---	---	---	----	----	----	----	----	----	----	----

$z = 19$



Suchalgorithmen,  
Aufwandsabschätzungen

# Suchproblem / Sequentielle Suche

```
public class Zahlenfolge {
```

```
// wie zuvor
```

```
public boolean suchen(int z) {  
    for (int zahl : a) {  
        if (zahl == z) {  
            return true;  
        } else if (z < zahl) { // breche Suche ab  
            return false;  
        }  
    }  
  
    return false;  
}
```

**Ist diese Variante schneller / besser?**



# Suchproblem / Zeitaufwand

- Zeitmessen umständlich und aufwendig
  1. Algorithmus muss implementiert werden
  2. Gemessene Zeit hängt von konkretem Computer ab (CPU, Auslastung, RAM, usw. )
- Wie kann man Algorithmen vergleichen?
  - Problemgröße fixieren: in eine Maßzahl fassen
    - $n$  = Anzahl Zahlen in denen gesucht wird
    - $n$  Größe Schachbrett bei Springerproblem, usw.)
- Wie kann man die „Güte“ eines Algorithmus *ohne* Zeitmessen in Abhängigkeit von  $n$  bestimmen?



# Inhalt

- Suchalgorithmen
  - Sequentielle Suche
  - Zeitaufwand, Aufwandsabschätzung
  - Binärsuche



# Zeitaufwand

- **Ziel: Zeitaufwand schätzen, statt messen**
  - Ausgeführte Anweisungen Algorithmus *grob* zählen
  - Genaues Zählen irrelevant
  - Iterationen / Rekursion: Wiederholungen schätzen
- **Schätzungen als Funktion von n betrachten**
  - $T(n) = \dots$  / T wie Time
  - Gibt an, wie viele Einzelschritte ein Algorithmus bei einem Problem der Größe n ungefähr ausführt
- **Schätzung hängt auch von der Eingabe ab**
  - Unterscheiden zwischen „guten“ und „schlechten“ Eingaben der Größe n



# Zeitaufwand

- Berechenbarkeitsmodell
  - Vorgabe einer bestimmten CPU
- Sei  $P$  ein Problem und  $A$  ein Algorithmus, der  $P$  löst und auf der CPU abläuft
  - $T(A,P)$  sei Anzahl der Befehle, die insgesamt ausgeführt werden, um  $P$  mit  $A$  zu lösen
- Es interessieren nur Probleme  $P$ , mit einer bestimmten Größe ( $|P| = n$ )
  - $n$  hängt vom Problem am, z.B. Anzahl zu durchsuchender Elemente oder Größe des Schachbretts
- Messen nötig (zählen), statt nur schätzen



# Zeitaufwand

## 1. Günstigster Fall (best case)

- Problem, bei dem der Algorithmus A am wenigsten Einzelschritte benötigt

$$T_{bc}(n) := \min \{T(A, P) : |P| = n\}$$

## 2. Schlechtester Fall (worst case)

- Problem, bei dem der Algorithmus die meisten Einzelschritte benötigt

$$T_{wc}(n) := \max \{T(A, P) : |P| = n\}$$

## 3. Mittlerer Fall (average case)

$$T_{ac}(n) := \frac{\sum_{|P|=n} T(A, P)}{|\{P : |P| = n\}|}$$

# Zeitaufwand

- Schätzen (statt Messen)
- Im schlimmsten Fall:  $T(n) \sim n$ 
  - Gesucht Element befindet sich immer am Ende des Feldes
  - Schleife wird  $n$  mal durchlaufen
- Im günstigsten Fall:  $T(n) \sim 1$ 
  - Erste Element ist immer das gesuchte
  - Schleife wird sofort abgebrochen
- Im mittleren Fall:  $T(n) \sim n/2$

```
public boolean suchen(int z) {  
    for (int zahl : a) {  
        if (zahl == z) {  
            return true;  
        }  
    }  
  
    return false;  
}
```

Nächste Element holen  
kostet „konstante“ Zeit.

Bedingung kostet „konstante“ Zeit.

A

# Zeitaufwand

- Im schlimmsten Fall:  $T(n) \sim 2 * n$ 
  - Ca. doppelt so viele Anweisungen in for-Schleife werden verwendet
  - Gesucht Element befindet sich immer am Ende des Feldes
  - Schleife wird n mal durchlaufen
- Im günstigsten Fall:  $T(n) \sim 1$
- Im mittleren Fall:  $T(n) \sim n/2$

```
public boolean suchen(int z) {
    for (int zahl : a) {
        if (zahl == z) {
            return true;
        } else if (z < zahl) {
            return false;
        }
    }
}
```

Nächste Element holen  
kostet „konstante“ Zeit.

B

Bedingung kostet „konstante“ Zeit.

Ist A besser als B?

```
return false;
```

# Zeitaufwand

- O-Kalkül

- Genaues „Zählen“ der Befehle nicht nötig
- Funktionen, die sich nur gering unterscheiden, werden zusammengefasst
- *Etwa*:  $f(n)$  wächst höchstens so schnell wie  $g(n)$ 
  - Anzahl Befehle  $T(n)$  abschätzen
- Ziel sind dann Aussagen wie:
  - Zeit  $T(n)$  des Algorithmus ist mindestens so schnell wie  $g(n) = n \cdot n$  falls  $n$  sehr, sehr groß wird

$$\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c \quad O(g(n)) := \left\{ f(n) : \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c \right\}$$

$$\lim_{n \rightarrow \infty} \frac{n}{n^2} = 0$$

← Ungefähre Schätzung für schlimmsten Fall  
Durch Grenzwert fallen kleine Ungenauigkeiten weg



# Zeitaufwand

- O-Kalkül (Details Theoretische Informatik)
  - $O(1)$ : Menge aller konstanten Funktionen  
 $f(n) = -5, f(n) = 16$
  - $O(n)$ : Menge aller Funktionen, die linear sind oder langsamer  
 $f(n) = n, f(n) = 5, f(n) = n+7, f(n) = 2n-15$
  - $O(n^2)$ : Menge aller Funktionen, die quadratisch wachsen oder langsamer:  
 $f(n) = n^2, f(n) = n^2 + 7n, f(n) = n, f(n) = 17$
- Schreibweise:  $f(n) = O(g(n))$  statt  $f(n) \in O(g(n))$
- Zeitaufwand sequentielle Suche:
  - $O(n)$  im schlimmsten und mittleren Fall wird nicht überschritten
  - $O(1)$  im besten Fall wird nicht überschritten



# Zeitaufwand

- Algorithmen mit Zeitaufwand von höchstens  $O(n)$  sind „besser“ als  $O(n^2)$ 
  - Sie unterscheiden sich durch *mehr* als nur einen konstanten Faktor
- $O(n)$  Algorithmen mindestens etwa  $n$  mal schneller als  $O(n^2)$  Algorithmen

Größenordnung, die ein  $O(n)$  Algorithmus an Zeit für ein Problem der Größe  $n = 100$  verbraucht

	$n = 1$	$n = 10$	$n = 100$	$n = 1\ 000$	$N = 10\ 000$
$O(n)$	1	10	100	1 000	10 000
$O(n^2)$	1	100	10 000	1 000 000	100 000 000



# Zeitaufwand

- Bestimmen des Zeitaufwands eines Programms
  - Bestimme Zeitaufwand einzelner überschaubarer Programmfragmente
    - „int i = j + 2;“ hat z.B. konstanten Zeitaufwand  $O(1)$
  - Bei sequentiellen Programmteilen:  
Gesamtaufwand ist Summe der einzelnen Aufwände
  - Bei Iterationen / Schleifen
    - Schätze ab, wie oft iteriert wird
    - Gesamtaufwand ist Aufwand des Schleifenrumpfs multipliziert mit Anzahl Durchläufen
  - Vereinfache Ausdrücke wann immer möglich



# Zeitaufwand

Zeitaufwand	Fragmente	Beispiele
O(1)	Deklaration (keine init. Felder) Wertzuweisung Zugriff Feldelement Ausdruck mit Werten	<pre> int i; Person egon; int a[]; i = 0; egon = null; personen[i] matrix[x][y] (i &lt; 0 &amp;&amp; egon != null &amp;&amp; personen[i] == egon) i++ </pre>



# Zeitaufwand

- Bei Feldern
  - Aufwand hängt von Länge des Feldes ab
  - Länge hängt von  $n$  ab ( $f(n) = n$ ,  $f(n) = n + 10$ )
- Aufwand  $O(f(n))$

Zeitaufwand	Fragmente	Beispiele
$O(n)$	Deklaration Felder  Feldobjekte anlegen	<pre>new int[n]; new Person[n]; new int[n+10]; new int[n] new Person[n]</pre>
$O(n^2)$		<pre>new int[n][n+2]</pre>



# Zeitaufwand

## Rechenregeln für O-Kalkül (siehe Theoretische Informatik)

1.  $O(g(n)) + O(f(n)) = O(g(n) + f(n))$   
(Hintereinanderausführen zweier Programme)  
 $O(n) + O(n^2) = O(n + n^2)$
2.  $O(g(n)) \cdot O(f(n)) = O(g(n) \cdot f(n))$   
(Iteration)  
 $O(n) \cdot O(n) = O(n^2)$
3.  $O(c \cdot f(n)) = O(f(n))$  für Konstante  $c$
4.  $O(c) = O(1)$   
(Spezialfall von 3 für  $f(n) = 1$ )
5.  $O(g(n) + f(n)) = O(g(n))$  falls  $f(n) = O(g(n))$   
( $g$  wächst asymptotisch mindestens so schnell wie  $f$ )  
 $O(n^2 + n) = O(n^2)$



# Zeitaufwand

- Bei while Schleifen
  - Aufwand  $O(f(n))$  des Rumpfs der Schleife bestimmen
  - Aufwand  $O(g(n))$  Bedingung
  - Anzahl Durchläufe  $h(n)$  der Schleife in Abhängigkeit von  $n$  bestimmen
  - Gesamtaufwand ist  $O( h(n) \cdot ( f(n) + g(n) ) )$
- In Praxis: einfacher, da man oft „sieht“, welche Teile der Funktion relevant sind



# Zeitaufwand

- Bei if / switch
  - Aufwand  $O(f(n))$  der Bedingung ermitteln
  - Wahrscheinlichkeit  $p(n)$  in Abhängigkeit von  $n$ , das Bedingung eintritt ermitteln
  - Aufwand  $O(g(n))$  des zugehörigen Rumpfes ermitteln
  - Gesamtaufwand ist  $O(f(n) + p(n) \cdot g(n))$
  - Bei kaskadierenden if-else und switch entsprechend komplizierter
- In der Praxis: Oft Annahme, das  $p(n) = 1$  ist (schlimmster Fall)



# Zeitaufwand

- Bei Methoden
  - Aufwand des Methodenrumpfs in Abhängigkeit der Methodenparameter (und ggf. Objektattribute) bestimmen.
  - Aufwand eines Methodenaufrufs in Abhängigkeit der Parameterwerter bestimmen
- In der Praxis
  - Nur von Eingabegröße  $n$  abhängige Parameter und Attribute berücksichtigen
  - Nur Werte berücksichtigen, die bei Aufrufen überhaupt vorkommen.

# Zeitaufwand / Sequentielle Suche

Aufwand für **schlimmsten** Fall:

a enthält z überhaupt nicht, Schleife wird nicht durch return verlassen, also vollständig durchlaufen

```
public boolean suchen(int z) {
    for (int zahl : a) {
        if (zahl == z) {
            return true;
        }
    }
    return false;
}
```

$$O(n) * O(1) + O(1) = O(n*1 + 1) = O(n) = T_{wc}(n)$$

# Zeitaufwand / Sequentielle Suche

Aufwand für **besten** Fall:

a[0] enthält gesuchtes z, for-Schleife wird verlassen

```
public boolean suchen(int z) {  
    for (int zahl : a) {  
        if (zahl == z) {  
            return true;  
        }  
    }  
}
```

```
return false;  
}
```

$$O(1) * O(1) = O(1*1) = O(1)$$

# Zeitaufwand / Sequentielle Suche

Aufwand für **mittleren** Fall:

Schleife wird im Mittel bei der **Hälfte** der Durchläufe abgebrochen

```
public boolean suchen(int z) {  
    for (int zahl : a) {  
        if (zahl == z) {  
            return true;  
        }  
    }  
  
    return false;  
}
```

$$O(n/2) * O(1) = O(n/2 * 1) = O(n)$$



# Zeifaufwand / Sequentielle Suche

- A und B sind „gleich“ schnell
  - jeweils  $O(n)$  im schlimmsten und durchschnittlichsten Fall
  - $O(1)$  im besten Fall
- Gibt es bessere Suchverfahren?
  - Ja (später)



n	Falls a = b
10	A <sub>1</sub> ca. 10 mal schneller als A <sub>2</sub>
100	A <sub>1</sub> ca. 100 mal schneller als A <sub>2</sub>
1 000	A <sub>1</sub> ca. 1 000 mal schneller als A <sub>2</sub>
10 000	usw,

„ca.“ = A<sub>1</sub> und A<sub>2</sub> unterscheiden sich um einen konstante Faktor

```

int multiplizieren(int a, int b) {
    int produkt = 0;

    while ( b > 0 ) {
        produkt = produkt + a;
        b = b - 1;
    }

    return produkt;
}
    
```

**A<sub>1</sub>**

n = b

**O(n)**

```

int multiplizieren(int a, int b) {
    int produkt = 0;

    while ( b > 0 ) {
        for (int i = 0; i < a; i++) {
            produkt = produkt + 1;
        }
        b = b - 1;
    }

    return produkt;
}
    
```

**A<sub>2</sub>**

n = max(a,b)

**O(n<sup>2</sup>)**



# Zeitaufwand

## Zeitaufwand rekursiver Algorithmen?

```
public void hanoi(int n, int von, int zu, int hilfe) {  
    if (n == 1) {  
        System.out.println(n + ": " + von + "-->" + zu);  
    } else {  
        hanoi(n - 1, von, hilfe, zu);  
        System.out.println(n + ": " + von + "-->" + zu);  
        hanoi(n - 1, hilfe, zu, von);  
    }  
}
```



# Zeitaufwand / Hanoi

## Rekurrenzgleichung

```
void hanoi(int n, int von, int zu, int hilfe) {  
    if (n == 1) {  
        System.out.println(n + ": " + von + "-->" + zu);  
    } else {  
        hanoi(n - 1, von, hilfe, zu);  
        System.out.println(n + ": " + von + "-->" + zu);  
        hanoi(n - 1, hilfe, zu, von);  
    }  
}
```

$$T(n) = c \quad , \text{ falls } n = 1$$

$$T(n) = T(n-1) + c + T(n-1) \quad , \text{ falls } n > 1$$
$$= 2 T(n-1) + c$$



# Zeitaufwand / Hanoi

$$\begin{aligned}T(n) &= 2T(n-1) + c \\&= 2 \cdot (2 \cdot T(n-2) + c) + c \\&= 4T(n-2) + 2c + c \\&= 2(4T(n-3) + c) + 2c + c \\&= 8T(n-3) + 4c + 2c + c \\&= 2^4 T(n-4) + 2^3 c + 2^2 c + 2^1 c + 2^0 c \\&= \dots \\&= 2^{n-1} T(1) + 2^n c + \dots + c \\&= 2^n c - 1 = O(2^n)\end{aligned}$$

# Zeitaufwand / Fakultät

$$fak(n) := \begin{cases} 1 & n = 0 \\ fak(n-1) \cdot n & n > 0 \end{cases}$$

```
public long fak(long n) {
    if (n == 0) {
        return 1;
    } else {
        return fak(n-1) * n;
    }
}
```

$$\begin{aligned} T(0) &= c \\ T(n) &= T(n-1) + c \\ &= T(n-2) + c + c \\ &= T(n-3) + 3 \cdot c \\ &= \dots \\ &= T(0) + n \cdot c \\ &= (n+1) \cdot c \\ &= O(n+1) = O(n) \end{aligned}$$



# Zeitaufwand / Rekurrenzgleichungen

- Rechenregeln für O-Kalkül gelten *nicht* in Rekurrenzgleichungen:

$$T(n) = T(n-1) + c$$

$$\neq O(T(n-1) + c)$$

$$\neq O(T(n-2) + c + c)$$

$$\neq O(T(n-2)) + O(c + c)$$

$$\neq \dots$$

$$\neq O(1)$$



# Inhalt

- Suchalgorithmen
  - Sequentielle Suche
  - Zeitaufwand, Aufwandsabschätzung
  - **Binärsuche**



# Suchproblem / Sequentielle Suche

- Sequentielle Suche für viele Probleme zu langsam
- Beschleunigen, durch *Vorverarbeitung*, damit Suche gezielter und früher abgebrochen wird
- Analog: Alphabetische Ordnung von Akten
  - Müller bei „M“ anfangen zu suchen, nicht bei „A“
  - Müller mehr am Ende der Akten innerhalb von „M“ suchen
  - Suche abbrechen, wenn wir zum Beispiel bei „Myers“ angekommen sind

# Suchproblem / Sequentielle Suche

13	4	12	-23	11	8	12	5	21	8	-6	19
----	---	----	-----	----	---	----	---	----	---	----	----

- Zahlenfolge in geordnete Reihenfolge bringen
  - Wenn  $a[i] > z$  ist, kann lineare Suche abgebrochen werden

-23	-6	4	5	8	8	11	12	12	13	19	21
-----	----	---	---	---	---	----	----	----	----	----	----

$z = -9837$       Abbruch bei erstem Element, da  $z < -23$

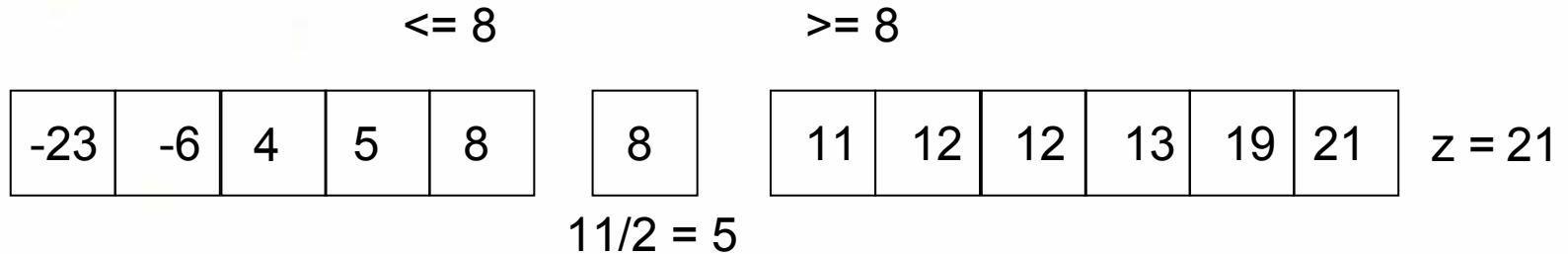
$z = 7$       Abbruch beim 5. Element, da  $z < 9$

$z = 21$       Abbruch erst beim letztem Element

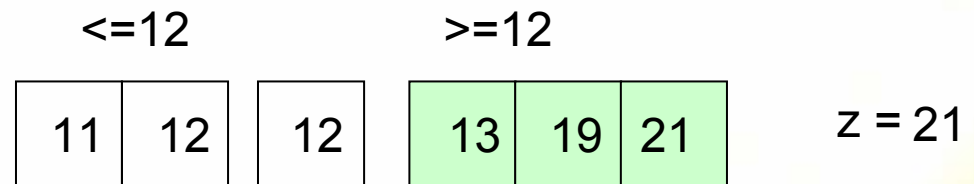
- Im *schlimmsten* Fall weiterhin Zeitaufwand  $O(n)$ :  
keine signifikante Verbesserung

# Suchproblem / Binärsuche

- Suche nicht vorne beginnen, sondern bei  $a[n/2]$

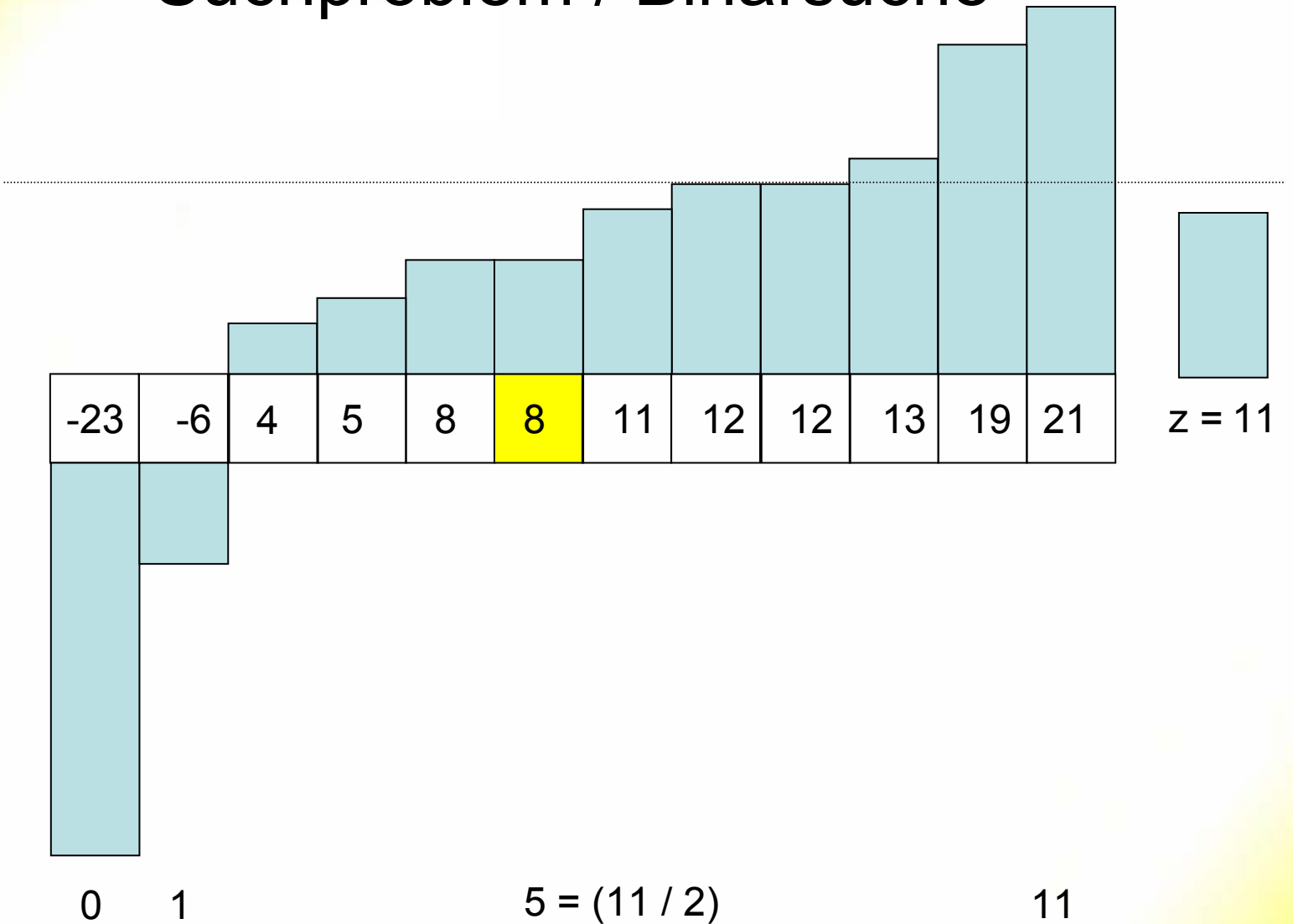


- Falls  $a[n/2] \neq z$ , dann entweder links oder rechts weitersuchen, je nachdem  $z$  kleiner gleich oder größer gleich  $a[n/2]$



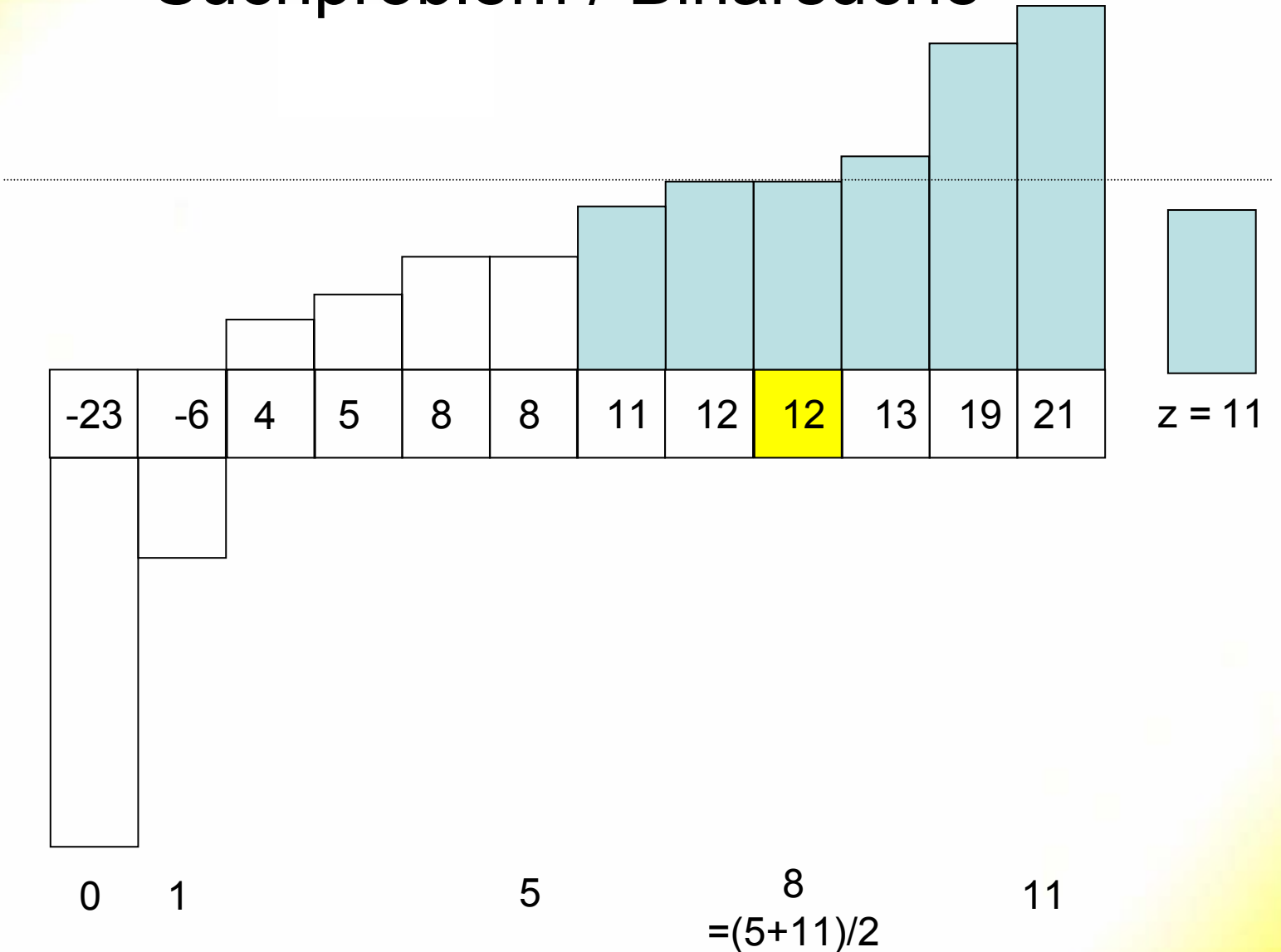


# Suchproblem / Binärsuche



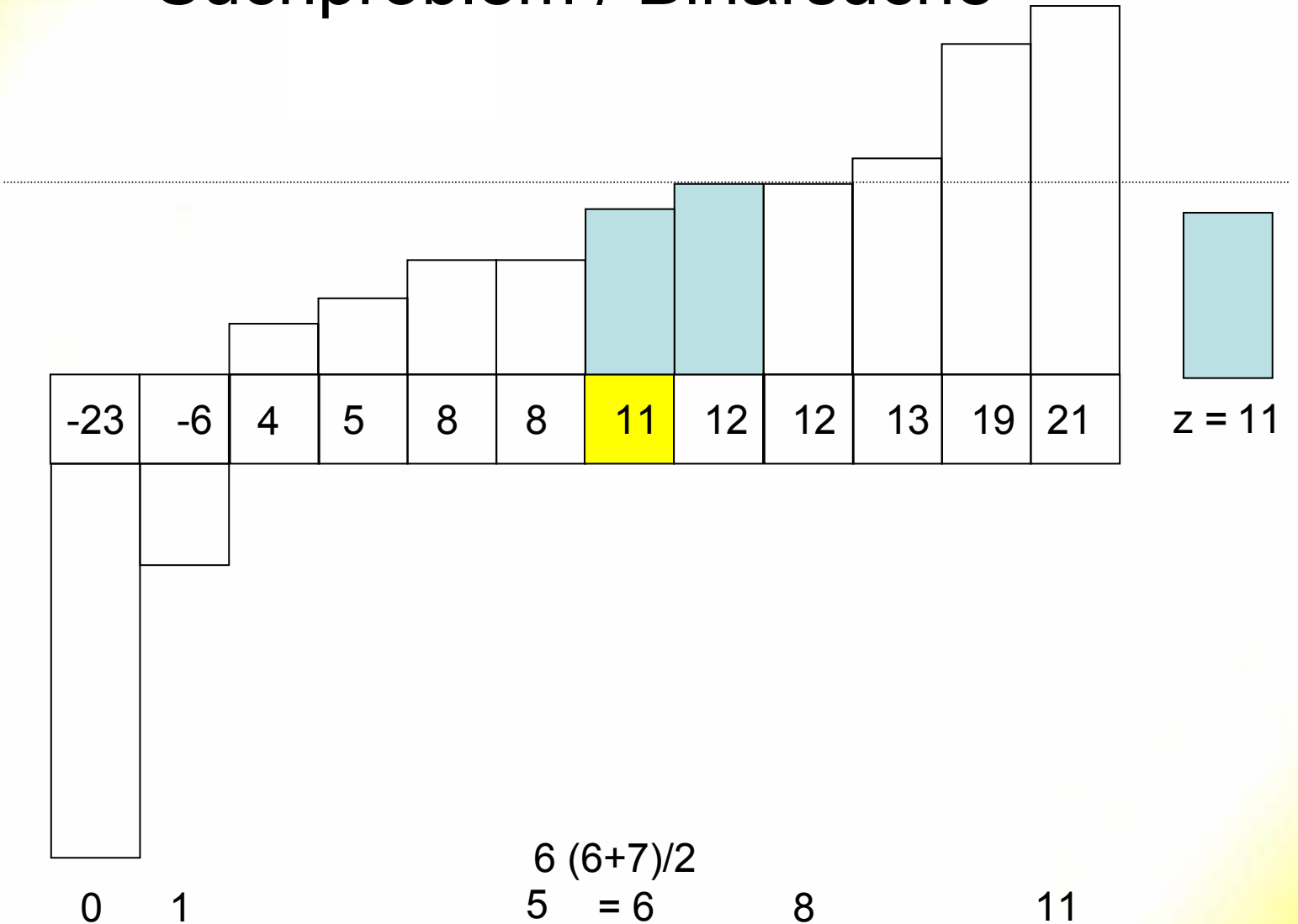


# Suchproblem / Binärsuche





# Suchproblem / Binärsuche





# Suchproblem / Binärsuche

-23	-6	4	5	8	8	11	12	12	13	19	21
-----	----	---	---	---	---	----	----	----	----	----	----

$z = 5$  suchen

$z = 13$  suchen



# Suchproblem / Binärsuche

- Algorithmus `boolean binaerSuchen(z, links, rechts)`
- Parameter:
  - zu suchende Element `z`
  - linke Grenze `links`
  - rechte Grenze `rechts`
- Falls `links > rechts` gib `false` zurück
- Berechne `mitte = (links+rechts) / 2`
- Falls `a[mitte] == z` gib `true` zurück
- Falls `a[mitte] < z` gib `binaerSuchen(z, mitte+1, rechts)` zurück
- Falls `a[mitte] > z` gib `binaerSuchen(z, links, mitte-1)` zurück

# Suchproblem / Binärsuche

...

```
public boolean suchen(int z) {  
    return binaerSuchen(z, 0, a.length - 1);  
}  
  
private boolean binaerSuchen(int z, int links, int rechts) {  
    int mitte = ( links + rechts) / 2;  
    if (links > rechts) {  
        return false;  
    } else if ( a[mitte] == z ) {  
        return true;  
    } else if ( z > a[mitte] ) {  
        return binaerSuchen(z, mitte+1, rechts);  
    } else {  
        return binaerSuchen(z, links, mitte-1);  
    }  
}
```



- $a[] = \{0,0,0,0,0, \dots, 0,12\}$
- $z = 12$
- $n = a.length = 1\ 000\ 000$
  
- Höchstens 20 Schritte
  
- Wie hoch ist der Zeitaufwand für Binärsuche?
  - Pro Aufruf Zeit beschränkt durch Konstante, da keine Schleifen im Rumpf vorhanden
  - Ordnung Zeitaufwand gleich Anzahl rekursiver Aufrufe

$l = 0, r = 999999$   
 $l = 500000, r = 999999$   
 $l = 750000, r = 999999$   
 $l = 875000, r = 999999$   
 $l = 937500, r = 999999$   
 $l = 968750, r = 999999$   
 $l = 984375, r = 999999$   
 $l = 992188, r = 999999$   
 $l = 996094, r = 999999$   
 $l = 998047, r = 999999$   
 $l = 999024, r = 999999$   
 $l = 999512, r = 999999$   
 $l = 999756, r = 999999$   
 $l = 999878, r = 999999$   
 $l = 999939, r = 999999$   
 $l = 999970, r = 999999$   
 $l = 999985, r = 999999$   
 $l = 999993, r = 999999$   
 $l = 999997, r = 999999$   
 $l = 999999, r = 999999$



# Suchproblem / Binärsuche

## Rekurrenzgleichung

- $T_{wc}(n) = c$  für links > rechts (z nicht gefunden)
- $T_{wc}(n) = T_{wc}(n/2) + c$  (linke/rechte Hälfte)

---


$$\begin{aligned}
 T(n) &= T(n/2) + c \\
 &= T(n/4) + c + c \\
 &= T(n/8) + c + c + c \\
 &= T(n/2^3) + 3c \\
 &= \dots \\
 &= T(1) + kc \\
 &= T(0) + c + kc \\
 &= c + c + kc \\
 &\stackrel{n=2^k, k=\log_2 n}{=} 2c + \log_2(n)c = O(\log_2 n)
 \end{aligned}$$



# Suchproblem / Binärsuche

- $n = 2^k$  insgesamt  $k+2$  rekursive Aufrufe  
 $k = 2 + \log_2 n$  rekursive Aufrufe
- Zeitaufwand  $T_{wc}(n) = O(\log_2 n)$
- 10 Schritte bei 1 000 Zahlen ( $1024 = 2^{10}$ )
- 20 Schritte bei 1 000 000 Zahlen (ca.  $2^{20}$ )
- 27 Schritte bei 100 000 000 Zahlen (ca.  $2^{27}$ )
- 40 Schritte bei 1 000 000 000 000 Zahlen (ca.  $2^{40}$ )
- Zeitabschätzung:
  - ca. ein Dutzend Maschinenbefehle pro rekursiven Aufruf
  - 1 GHz: 1ns pro Maschinenbefehl
  - 10 ns pro rekursivem Aufruf
  - 270 ns bei 100 000 000
  - 270 ms bei 1 000 000 mal 100 Millionen Zahlen



# Suchproblem / Binärsuche

- $O(\log_2 n)$  besser als  $O(n)$ , da  $n \neq O(\log_2 n)$
- Wie schnell ist  $O(\log_2 n)$  in Realität?
- 1 Mio x 100 Mio Zahlen suchen (1,7 GHz, P4M)
  - Binärsuche: 430 ms (nach Schätzung zuvor 270 ms)
  - Sequentielle Suche: ca. 10 min
- Suche von Zeichenketten/komplexe Objekte ist 10 bis 100 langsamer
  - 1 000 mal 1 Mio Zeichenketten: ca. 2 h bei sequentieller Suche
  - Zu langsam für die Praxis



# Zeitaufwand und Zeit

Zeitaufwand	10	1 000	10 000	100 000	10 Mio	100 Mio	10 000 Mio
<b><math>O(\log_2 n)</math></b> Binärsuche	praktikabel					420 ns	
<b><math>O(n)</math></b> Sequentielle Suche				< ms	0,070s	1 s	1 000 s
<b><math>O(n \log_2 n)</math></b> Sieb des Eratosthenes					1,5 s	17,65 s	
<b><math>O(n^2)</math></b> Bubblesort (später)		10 ms	1s	2 min	Tage, Wochen		
<b><math>O(2^n)</math></b> Backtracking, Fibonacci rekursiv	ms	Wochen Jahre	zu lang			Nicht mehr praktikabel	

3 s bei  $n = 40$   
1 min bei  $n = 46$