

Vorlesung 7

- Algorithmen
 - Definition
 - 1. Beispiel von Algorithmus: Lineare Suche
 - Iterativer Algorithmus
 - Struktogramm
 - Effizienz eines Algorithmus
 - Rechenschritt
 - Aufwand (Komplexität)
 - Aufwandseinstufung (O-Notation)
 - Beispiele:
 - Matrixmultiplikation
 - Binäre Suche

Effizientes Programmieren: Algorithmen

-*Algorithmen*: Verfahren, dass verwendet wird, um ein Problem durch eine endliche Anzahl von Schritten zu lösen.

Algorithmus: Bestandteile

Die Bestandteile ein Algorithmus sind:

- **Sachen (Information)**, die durch den Algorithmus bearbeiten werden.

Variablen

- **Operationen**, die auf die Sachen ausgeführt werden.

Operatoren-Funktionen

- **Arbeitsreihfolge**, in der die Operationen ausgeführt werden.

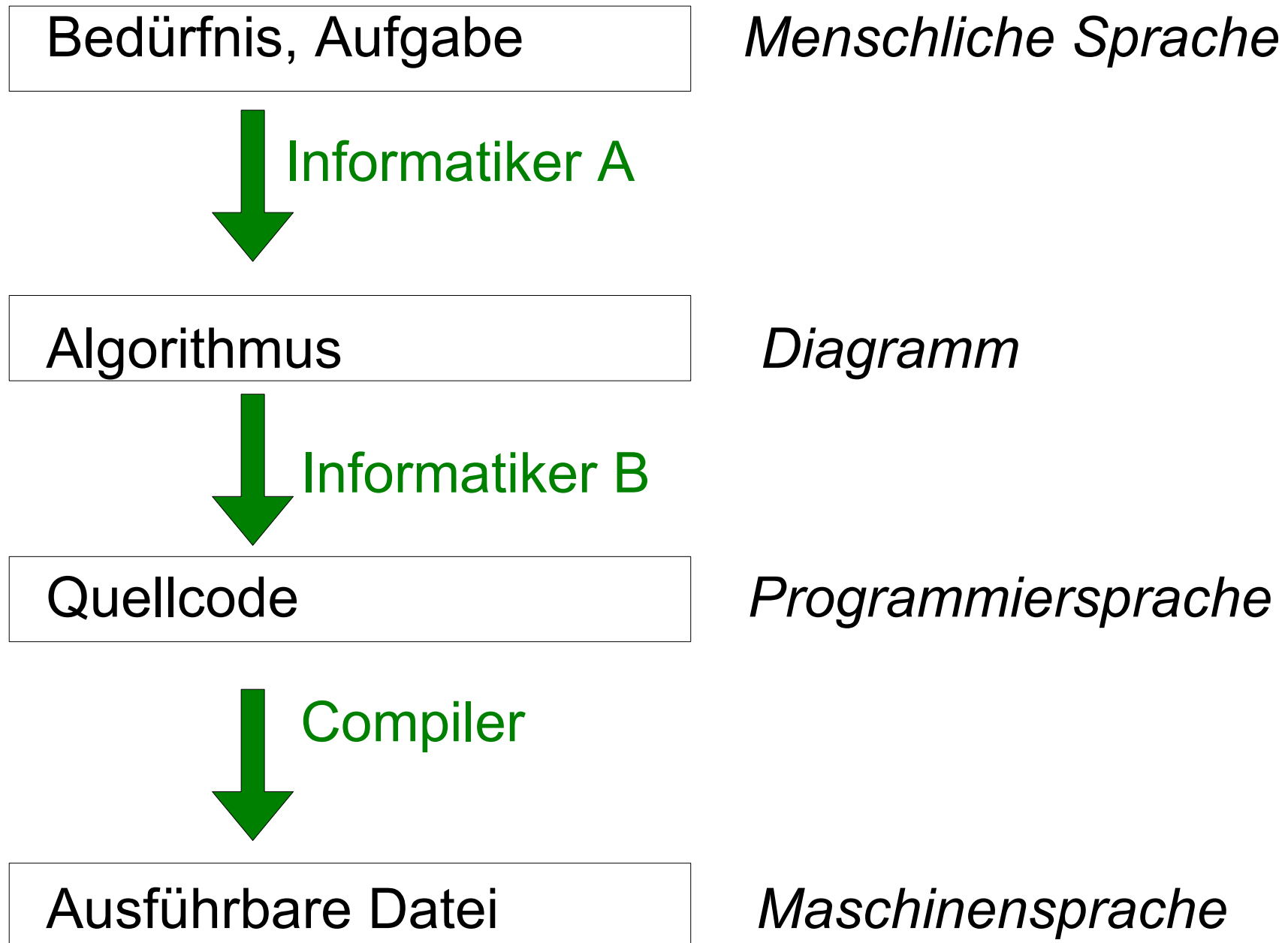
Kontrollstrukturen-Schleifen

Algorithmus - Programm

Wenn ein Algorithmus so formuliert ist, dass ein **Prozessor** ihn ausführen kann, dann wird auch **Programm** genannt.

Das heißt, jedes Computerprogramm ist ein Algorithmus. Jedoch nicht alle Algorithmen müssen ein Programm sein.

Aufgabe - Algorithmus - Programm



Algorithmen: Suche

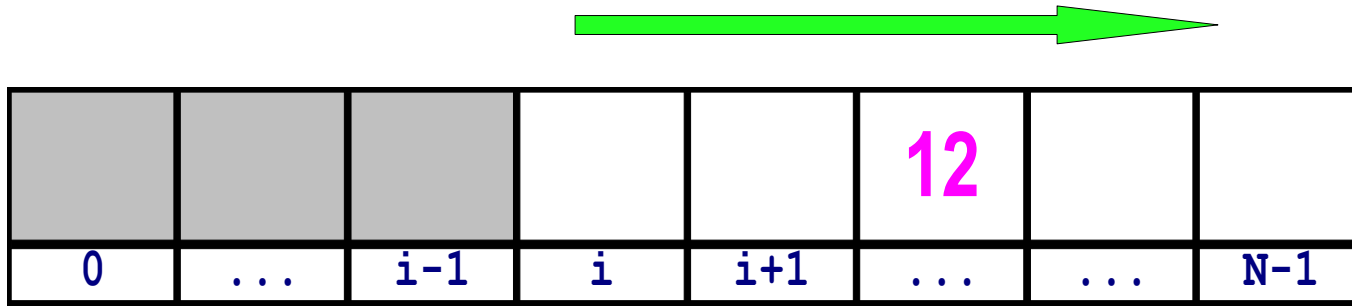
Aufgabe:

- Es wird ein Wert X (zum Beispiel $X=12$) in einem Feld (Array) mit n Elementen (von 0 bis $(n-1)$) gesucht.

20	1	-9	7	100	12	-20	1
0	1	2	3	4	5	6	7

Lineare Suche: ein Iterativer Algorithmus

- **Annahme:** Das gesuchte Element befindet sich nicht bis auf einer gewissen Position i des Arrays



- **Iterativer Schritt:** Die Position i wird untersucht
 - Element gefunden: ENDE des Algorithmus
 - Element nicht gefunden, daher neue Annahme: „Das Element befindet sich ab Position $i+1$ des Arrays“
- **Anfangsannahme:** Das gesuchte Element befindet sich ab der Position NULL des Arrays
- **Endbedingung:** Wenn die Position n erreicht wird, ist das Element nicht im Vektor

Iterativer Algorithmus

- **Annahme:** Notwendige Voraussetzung für die Anwendbarkeit des Algorithmus.
- **Iterativer Schritt:** Verfahren die basiert auf die Annahme, das das zu lösende Problem verringert.
- **Anfangsannahme:** Voraussetzung um überhaupt mit dem Algorithmus anfangen zu können.
- **Endbedingung:** Bedingung, die ein unendlichen Algorithmus verhindert.

Algorithmen: Suche

Lösungsweg 1: die lineare Suche untersucht die Arrayelemente der Reihe nach bis das Element X gefunden ist.

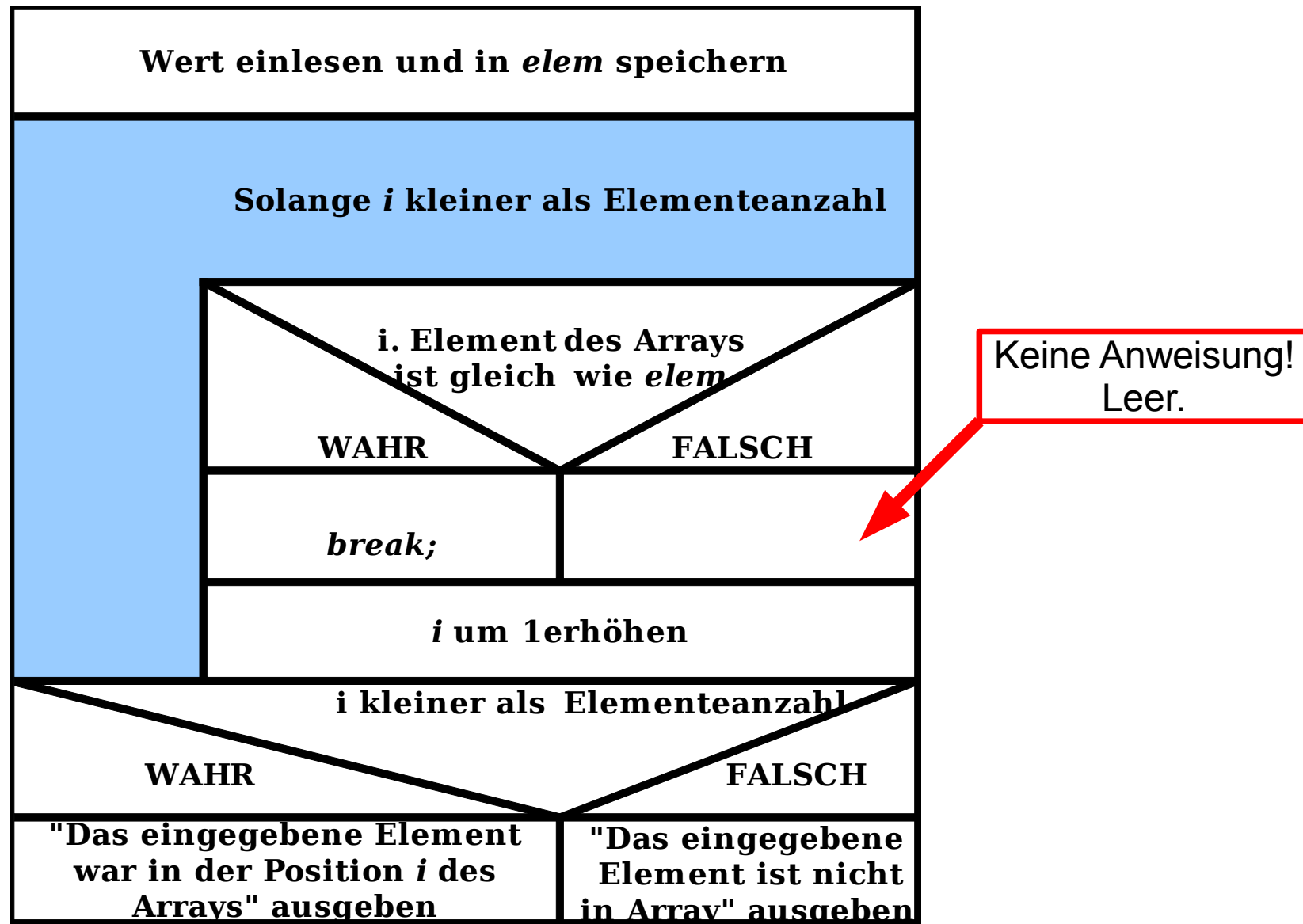
20	1	-9	7	100	12	-20	1
0	1	2	3	4	5	6	7

Darstellung des Algorithmus

Aus dem ersten Semester:kennen wir:

- *Struktogramme.*
- *Flussdiagramme.*

Diagramm-Darstellung: lineare Suche



Quellcode: lineare Suche

- Lineare Suche:

```
...  
element = 12; //gesuchtes Element  
  
for ( i = 0; i < n; i = i + 1 )  
{  
    if ( array [i] == element )  
        break; //Gefunden! Schleife abgebrochen.  
}  
...
```

Code: *LineareSuche.cpp*

Kontrollbefehle: *break*

- *break* bewirkt das sofortige Verlassen der entsprechenden "switch "- oder Schleife
- **Vorsicht!** In verschachtelten Schleifen wird durch *break* nur die unmittelbar umgebende Schleife verlassen.

Algorithmus: EFFIZIENZ

Die Effizienz: Verhalten von Algorithmen bezüglich Ressourcenbedarfs

- **Rechnungsbedarf:** Anzahl von nötigen Rechenschritte.
- **Speicherplatzbedarf:** Menge von nötigen Speicherplatz.

Algorithmus: EFFIZIENZ

RECHENSCHRITT

Ein **Rechenschritt** wird betrachtet als eine grundlegende Operation des Computers:

- ~~Abruf des Wertes einer Variable~~
(Um Einfachheit willen werden wir dies nicht als Rechenschritt betrachten)
- Zuweisung =
(Zugriff auf Speicher)
- Berechnung +, %, &&, ...
(Operation in Prozessor)
- ~~Aufruf funktion(,)~~
(kein Rechenschritt!
Besteht nämlich aus mehrere Schritte: alle ihre Anweisungen)

Algorithmus: EFFIZIENZ

- Im Allgemein den **Speicherplatzbedarf** kann immer abgedeckt werden, bei der Besorgung mehr Speicherplatzes.
 - Marktpreise für RAM ≈ 10 €/GB
 - Marktpreise für Festplattenspeicher ≈ 0.1 €/GB
- Der **Rechnungsbedarf** kann aber bei vielen Algorithmen nicht in einer vernünftigen Zeit erledigt werden. (10 Jahre, 1 Jahrhundert oder sogar mehr).
Die höchste Taktfrequenz der kommerziellen Prozessoren ist ≈ 3 GHz

Algorithmus: EFFIZIENZ

- Also die **Effizienz** eines Algorithmus wird wesentlich bestimmt von dessen **Rechenschritten**.
- Die Zeitaufwand eines Algorithmus wird proportional zu den Rechenschritten.
Vorausgesetzt, dass alle Rechenschritten werden gleich schnell erledigt.
 - Zeitaufwand \sim Rechenschritte

(rechnerische) KOMPLEXITÄT

Komplexität (Aufwand) Das Gebiet der Informatik, das die Effizienz der Algorithmen abschätzt.

Ähnliche Begriffe sind:

- Komplexität
- rechnerische Komplexität
- Aufwand
- Zeitaufwand

KOMPLEXITÄT:Beispiel

Aufgabe:

Die Folge **0,5,10,....., 1000** erzeugen

2 Mögliche Lösungen:

- Lösung A:

```
for (i=0; i<= 1000; i = i + 1)
{
    if (i % 5 == 0)
        cout << i << " ";
}
```

2 Mögliche Lösungen:

- Lösung B:

```
for (i=0; i<= 1000; i = i + 5)
{
    cout << i << " ";
}
```

Lösungsvergleich

- **Lösung A:**

1000 Schleifenvergleiche $i \leq 1000$
2 x 1000 Erhöhung und Zuweisung $i = i + 1$
+1000 Vergleiche $i \% 5 == 0$
200 Ausgaben *cout*

4200 Rechenschritten

- **Lösung B:**

200 Schleifenvergleiche $i \leq 1000$
2 x 200 Erhöhung und Zuweisung $i = i + 5$
+200 Ausgaben *cout*

800 Rechenschritten

- **Fazit:**

Lösung B ist ungefähr 5 mal schneller als Lösung A.
Das Ergebnis von beiden Lösungen ist aber die gleiche Ausgabe.

KOMPLEXITÄT:Beispiel

Allgemeiner Fall: Folge bis N

Aufgabe:

Die Folge **0,5,10,.....N-5, N** erzeugen

Lösungsvergleich

- Größe der Folge ist N

- Lösung A:

N Schleifenvergleiche $i \leq 1000$

$2 * N$ Erhöhung und Zuweisung $i = i + 1$

$+ N$ Vergleiche $i \% 5 == 0$

$N/5$ Ausgaben cout

$4,2 * N$ Rechenschritten

- Lösung B:

$N/5$ Schleifenvergleiche $i \leq 1000$

$2 * N/5$ Erhöhung und Zuweisung $i = i + 5$

$+ N/5$ Ausgaben cout

$0,8 * N$ Rechenschritten

Lösungsvergleich

- Größe der Folge ist N
- Lösung A:
4,2*N Rechenschritten
- Lösung B:
0,8*N Rechenschritten
- **Fazit:**
Lösung B ist ungefähr 5 mal schneller als Lösung A.
Alle Lösungen sind doch LINEAR. Proportional zu N
 - Lösung A: Zeitaufwand (N) $\sim N$
 - Lösung B: Zeitaufwand (N) $\sim N$

KOMPLEXITÄT

Allgemeiner Fall: Folge bis N

Die Komplexität (Zeitaufwand) ist eine Funktion der eingegebenen Elemente $T(N)$.

Normalerweise wird nicht die genaue Funktion ausgedrückt aber den wichtigsten Glied für großes N .

Beispiel:


$$T(N) = 5 \cdot N^3 + 8 \cdot N^2 + \log(N)$$

Das wird so ausgedrückt $T(N) = O(N^3)$

KOMPLEXITÄT

Einstufung der Komplexität

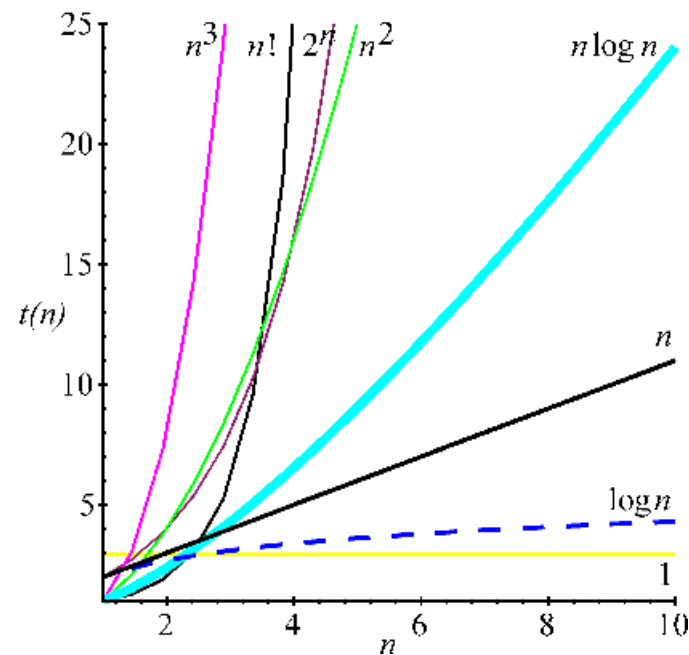
Komplexität als eine Funktion der eingegebenen Elemente. O Notation

- 
- $O(1)$: konstanter Aufwand (z. B. Summe der 3 letzten Arrayelemente),
 - $O(\log N)$: logarithmischer Aufwand (z. B. allgemeine Suchverfahren)
 - $O(N)$: linearer Aufwand (z. B. syntaktische Programmanalyse)
 - $O(N \log N)$: quasilinearer Aufwand (z. B. Sortierung)
 - $O(N^2)$: quadratischer Aufwand (z. B. Vektormultiplikation)
 - $O(N^k)$: polynomialer Aufwand (z. B. Matrizenmultiplikation (N^3)),

rechnerische KOMPLEXITÄT

Zeitaufwand $t(n)$ als eine Funktion der eingegebenen Elemente n .

Beispiel für kleine Werte von n (Größe der Eingabe)



Algorithmen (lineare Suche)

- Lineare Suche:

```
...
element = 5; //gesuchtes Element

for ( i = 0; i < n; i = i + 1 )
{
    if ( array [i] == element )
        break; //Gefunden! Schleife abgebrochen.
}
...
```

Bei jedem Ablauf der Schleife werden 4 Operationen (Rechenschritte) ausgeführt

- Vergleich $i < n$
- Vergleich `array[i] == element`
- Summe und Zuweisung von $i = i+1$

Wieviel mal wird die Schleife ausgeführt?

- Bester Fall (das Element in der ersten Position gefunden): 1 Mal
- Schlimmster Fall (das Element in der letzten Position gefunden): N Male
- Durchschnitt: $N/2$ Male

Algorithmen (lineare Suche)

- Komplexität

- *Bestes*: 1 mal Schleife * 4 Operationen = 4 Operationen
- *Schlimmstes*: N mal Schleife * 4 Oper. = 4N Operationen
- *Durchschnitt*: N/2 mal * 4 Oper = 2N Operationen

Dass heißt als Durchschnitt werden etwas **proportional** zu N
die Anzahl der Elemente

$$T(N) = O(N)$$

Schätzung der Zeit aus der Komplexität

- Durch den Aufwand $O(f(N))$ lassen Sie sich Zeit vergleichen.

– Beispiel:

Wenn die ein Algorithmus ist $O(N^2)$,
Die Zeit für N ist ungefähr $T(N) \approx \alpha N^2$

Dann die Zeit für $10 \cdot N$ ist $T(10 \cdot N) = \alpha(10N)^2 = 10^2 \alpha N^2 = 10^2 T(N) = 100 T(N)$

Das heißt, für quadratische Algorithmen $O(N^2)$ wird 10 mal mehr Elemente ist 100 mal mehr Zeitaufwand

Beispiel: Matrizenmultiplikation

- Drei verschachtelte Schleife macht den Algorithmus zu $O(N^3)$,
Die Zeit für N ist ungefähr $T(N) = \alpha N^3$

Dann die Zeit für $10*N$ ist $T(10*N) = \alpha(10N)^3 = 10^3 \alpha N^3 = \mathbf{10^3} * T(N)$

Fazit: Wenn die Größe der Matrix $N \times N$ wird mit 10 Multipliziert, die Zeit Aufwand wird 1000 mal erhöht.

- Testen sie diese Behauptung in *Matrix.cpp*

Algorithmen (Suche)

- Lösungsweg 2: Binäre Suche des Elements 12
Vorbedingung: Sortierter Array!

- **Mittelement** des Arrays wird überprüft
 - Falls es das Gesuchte ist, ist alles zu Ende
 - Sonst wird es dank der Sortierung entschieden, welche **Hälfte des Arrays** man weiter untersucht .
- Nun wird die Suche eingeschränkt zu einem Unterarray.

-9	1	7	10	12	100	200	1307
1	2	3	4	5	6	7	8

Ausgewählte Hälfte des Arrays

Algorithmen (Suche)

- Iterationsschritt

Mittelement des vorherigen Unterarrays wird überprüft

- Falls es das Gesuchte ist, ist alles zu Ende
- Sonst wird es dank der Sortierung entschieden, welche **Hälfte des Unterarrays** man weiter untersucht .

Ausgewählte Hälfte

The diagram shows a sorted array of 8 elements. A yellow bracket above the array highlights the subarray from index 4 to index 6. The middle element of this subarray, at index 5, is highlighted in green. The entire array is highlighted with a green bracket below it, indicating it is the current 'Unterarray'.

-9	1	7	10	12	100	200	1307
1	2	3	4	5	6	7	8

Unterarray

Binäre Suche: Effizienz

- **Binäre Suche.**

Größe der Eingabe: Ein Array mit N Elementen, das heißt mit Länge $L = N$ Elementen

Die Länge des Intervalls L_k [Grenze unten, Grenze oben] wird nach jedem Durchlauf der Schleife durch 2 geteilt. Wir haben dann die Folge für die Länge des Intervalls in jeder Schleife.

$$0. \quad L_1 = L \qquad L_1 = L$$

$$1. \quad L_2 = L / 2^1 \qquad L_2 = L / 2$$

$$2. \quad L_3 = L / 2^2 \qquad L_3 = L / 4$$

$$k. \quad L_k = L / 2^{k-1}$$

Binäre Suche: Effizienz

Binäre Suche.

Ende der Suche:

Die Suche Endet, im schlimmsten Fall, wenn mein Suchintervall besteht aus nur ein Element. Das heißt $L_k = 1$

Nach der Formel oben bedeutet das:

$$1 = L / 2^{k-1}$$

$$2^{k-1} = L \quad \text{wo die Länge } L = N \text{ Elementen}$$

$k = \log_2(N) + 1$ Das ist die Anzahl der Schleifen im schlimmsten Fall.

Binäre Suche: Effizienz

- **Binäre Suche.**

Deshalb ist die Effizienz proportional zum Logarithmus der Anzahl der Menge. Das heißt **$O(\log(N))$** .

Suche: Binäre vs. Lineare

- **Lineare** Suche.

die Effizienz proportional zur Anzahl der Elemente.
Das heißt **$O(N)$** .

- **Binäre** Suche.

Die Effizienz ist proportional zum Logarithmus der Anzahl der Elemente.
Das heißt **$O(\log(N))$** .

Für kleine Menge Elemente kann es sein sogar, dass die lineare Suche schneller als die binäre. Für große Anzahl der Elemente die binäre ist unvergleichbar schneller.

Code: `SucheVergleich.cpp`

Zeitaufwand: Allgemeines Beispiel

- Nehmen wir 5 Algorithmen mit Aufwand 1 , $\log(N)$, N , N^2 , 2^N .
- Der Zeitaufwand eines Rechenschritts ist 10^{-6} Sekunden.

So wird die Zeitaufwand jedes Algorithmus in Bezug auf die Größe der Eingabe

n	1	$\log(n)$	n	n^2	2^n
10	1	1	10	100	0,01 Sek
100	1	2	100	10000	40196936841331500 Jahre
1000	1	3	1000	1 Sek	.
10000	1	4	0,1 Sek	1.67 Min	.