LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Einführung in die Informatik:
Programmierung und Softwareentwicklung
Wintersemester 2018/19

Kapitel 10: Komplexität von Algorithmen und Sortierverfahren

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 19. Dezember 2018

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11

Ziele

- Komplexität bezüglich Laufzeit und Speicherplatzbedarf
- Bestimmen der Komplexitäten von Algorithmen
- Asymptotische Notation und Komplexitätsklassen
- Sortieralgorithmen:
  Insertion Sort, Bubble Sort und Selection Sort
  (Quicksort wird in Kap. 11 behandelt)
- Komplexität von Sortieralgorithmen

Beispiel: Lineare Suche eines Elements in einem Array (1)

```
static boolean linSearch(int[] a, int e){
   for (int i = 0; i < a.length; i++){
     if (a[i] == e){return true;}
   }
   return false;
}
```

X	Durchschnitt von 100 Messungen in Millisekunden
10.000.000	2.82
20.000.000	5.47
40.000.000	10.93
80.000.000	21.71
160.000.000	43.89
320.000.000	92.64
640.000.000	194.48
1.280.000.000	412.15

Tabelle: Zeitmessung der Suche X in einem Array der Größe X gefüllt von 1 bis X.

Beachte: Solche Messungen sind

- rechnerabhängig
- abhängig davon, ob andere Programme zeitgleich laufen
- **ungeeignet**, um die Komplexität genau und ein für alle Mal zu bestimmen
- geeignet, um einen „Eindruck“ für die mögliche Komplexität zu gewinnen

Komplexität von Algorithmen

Wir unterscheiden den **Zeitbedarf** und den **Speicherplatzbedarf** eines Algorithmus.

Beides hängt ab von

- den verwendeten Datenstrukturen,
- den verwendeten algorithmischen Konzepten (z.B. Schleifen),
- der verwendeten Rechenanlage zur Ausführungszeit
  (davon abstrahieren wir im Folgenden).

Zeit- und Speicherplatzbedarf

Der **Zeitbedarf** eines Algorithmus errechnet sich aus dem Zeitaufwand für

- die Auswertung von Ausdrücken, einschl. Durchführung von Operationen,
- die Ausführung von Anweisungen,
- organisatorischen Berechnungen (davon abstrahieren wir im Folgenden).

Der **Speicherplatzbedarf** eines Algorithmus errechnet sich aus dem benötigten Speicher für

- lokale Variable (einschließlich formaler Parameter)
- Objekte (einschließlich Arrays) und deren Attribute,
- organisatorische Daten (davon abstrahieren wir im Folgenden).

Beispiel: Lineare Suche eines Elements in einem Array (2)

```
static boolean linSearch(int[] a, int e){
   for (int i = 0; i < a.length; i=i+1 i = i+1 i++ ){
     if ( a[i] == e){return true;}
   }
   return false;
}
```

Beispiele:

```
int a = {30, 7, 1, 15, 20, 13, 28, 25}
boolean b1 = linSearch(a,30); // Zeit?
boolean b2 = linSearch(a,7); // Zeit?
boolean b3 = linSearch(a,23); // Zeit?
```


Beispiel: Lineare Suche eines Elements in einem Array (3)

```
static boolean linSearch(int[] a, int e){
   for (int i = 0; i < a.length; i = i + 1 i++ ){
     if ( a[i] == e){return true;}
   }
   return false;
}
```

Sei $n = $ `a.length` und e bereits eine Zahl.
Wir schätzen den maximalen Zeitbedarf des Aufrufs `linSearch(a,e)` ab

- 1 für Zuweisung `i = 0`
- pro Schleifendurchlauf (n mal):
  - Für `i < a.length`: 1 Zugriff + 1 Vergleich
  - Für `i++`: 1 Operation `(i+1)` + 1 Zuweisung `i = ...`
  - Für `a[i] == e`: 1 Arrayzugriff + 1 Vergleich
- letzter Test `i < a.length`: 1 Arrayzugriff + 1 Vergleich
- Aufruf `return false`: 1 Return

Ergibt: $1 + n \cdot (1+1+1+1+1+1) + 2 + 1 = 6 \cdot n + 4$

Beispiel: Lineare Suche eines Elements (2)

```
static boolean linSearch(int[] a, int e){
    for (int i = 0; i < a.length; i++){
      if (a[i] == e){return true;}
    }
    return false;
 }
```

Speicherplatzbedarf

- n+1 (für a mit length)
- 1 für e
- 1 für i
- 1 für Ergebnis

Speicherplatzbedarf gesamt $= n$ +4
Speicherplatzbedarf zusätzlich zur Eingabe: 2
Speicherplatzbedarf für verschiedene Aufrufe von `linSearch`:

```
int[] a = {30, 7, 1, 15, 20, 13, 28, 25};
boolean b1 = linSearch(a, 30); // Speicherplaetze: 12
boolean b2 = linSearch(a, 23); // Speicherplaetze: 12
```

Komplexitätsanalyse

- Der Zeitbedarf und der Speicherplatzbedarf einer Methode hängt i.a. ab von der aktuellen Eingabe.
- Gegeben sei eine Methode `static type1 m(type2 x) {body}`

  Notation: $T_{\mathtt{m}}(e)$ = Zeitbedarf des Methodenaufrufs $\mathtt{m}(e)$
  $S_{\mathtt{m}}(e)$ = Speicherplatzbedarf des Methodenaufrufs $\mathtt{m}(e)$
- Meist ist man am **Skalierungsverhalten** eines Algorithmus interessiert: Wie hängen Zeit- und Speicherplatzbedarf von der **Größe n der Eingabe e** ab (z.B. von der Länge eines Arrays)?
- Der Algorithmus zum Suchen eines Elements in einem Array hat für **Arrays gleicher Länge unterschiedliche Kosten** bzgl. der Zeit.
- Um solche Unterschiede abschätzen zu können, unterscheidet man die **Komplexität im schlechtesten, mittleren und besten Fall** (engl. worst case, average case, best case complexity).

Komplexitätsarten

- Zeitkomplexität im

  schlechtesten Fall (worst case):
  $$T^w_{\mathtt{m}}(n) = \max\{T_{\mathtt{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$$
  mittleren Fall (average case):
  $$T^a_{\mathtt{m}}(n) = \text{Durchschnitt von } \{T_{\mathtt{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$$
  besten Fall (best case):
  $$T^b_{\mathtt{m}}(n) = \min\{T_{\mathtt{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$$
- Speicherplatzkomplexität im

  schlechtesten Fall (worst case):
  $$S^w_{\mathtt{m}}(n) = \max\{S_{\mathtt{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$$
  mittleren Fall (average case):
  $$S^a_{\mathtt{m}}(n) = \text{Durchschnitt von } \{S_{\mathtt{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$$
  besten Fall (best case):
  $$S^b_{\mathtt{m}}(n) = \min\{S_{\mathtt{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$$

Beispiel: Lineare Suche, Platzbedarf

```
static boolean linSearch(int[] a, int e){
   for (int i=0; i < a.length; i++) {
     if (a[i] == e) {return true;}
   }
   return false;
}
```

Als Größenmaß für die Eingabe a, e wählen wir die Länge n des Arrays a. (Für das Skalierungsverhalten ist die Größe von e hier nicht relevant)

Speicherplatzbedarf insgesamt:

$$S^w_{\mathtt{linSearch}}(n) = S^a_{\mathtt{linSearch}}(n) = S^b_{\mathtt{linSearch}}(n) = n + 4$$

Normalerweise rechnet man den Platz, den die Eingabe belegt, selbst nicht dazu (denn der Methodenaufruf **erzeugt** diesen Platz nicht)
Wir betrachten den **zusätzlichen** Speicherplatzbedarf:

$$S^w_{\mathtt{linSearch}}(n) = S^a_{\mathtt{linSearch}}(n) = S^b_{\mathtt{linSearch}}(n) = 2$$

Beispiel: Lineare Suche, Zeitbedarf

```
static boolean linSearch(int[] a, int e){
   for (int i=0; i < a.length; i++) {
     if (a[i] == e) {return true;}  }
   return false;                          }
```

Schlechtester Fall: $T^w_{\mathtt{linSearch}}(n) = 6n + 4$

Bester Fall: $T^b_{\mathtt{linSearch}}(n) = 6$

Durchschnittlicher Fall:

- Wir nehmen an, dass das Element nach 1 bis n Schleifendurchläufen gefunden wird, und dass jeder dieser Fälle gleichwahrscheinlich ist.
- Laufzeit bei j Schleifendurchläufen:
  $6 \cdot j$ (für die Schleifendurchläufe)
  $+$ 2 (für `i = 0;` und `return true`)
  $-$ 2 (für letzten Schleifendurchlauf erfolgt kein `i++`)

$$T^a_{\mathtt{linSearch}}(n) = \frac{\sum_{j=1}^{n} 6 \cdot j}{n} = 6\frac{(n+1)n}{2} \cdot \frac{1}{n} = 3n + 3$$

Größenordnung der Komplexität: Die $\mathcal{O}$-Notation

- Eine **exakte** Beschreibung des Zeit- und Speicherplatzbedarfs wird schnell zu kompliziert, um praktikabel zu sein.
- Die Komplexität der Funktionen $T^w(n)$, $T^a(n)$, $T^b(n)$, $S^w(n)$, $S^a(n)$ und $S^b(n)$ wird häufig nur **bis auf konstante Faktoren** untersucht.
- Sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion. Wir definieren $\mathcal{O}(f(n))$ als die Klasse aller Funktionen, die nicht wesentlich schneller wachsen als $f(n)$:

Definition $(\mathcal{O}(f(n)))$

Eine Funktion $g(n)$ ist in $\mathcal{O}(f(n))$, falls es Zahlen $c > 0$ und n_0 gibt, so dass $0 \leq g(n) \leq c \cdot f(n)$ für alle $n > n_0$.

D.h. die Funktion $g(n)$ wächst höchstens so schnell wie $f(n)$, abgesehen von einer linearen Skalierung von $f(n)$.

$\mathcal{O}$-Notation: Beispiele

Definition $(\mathcal{O}(f(n)))$

Eine Funktion $g(n)$ ist in $\mathcal{O}(f(n))$, falls es Zahlen $c > 0$ und n_0 gibt, so dass $0 \leq g(n) \leq c \cdot f(n)$ für alle $n > n_0$.

Beispiele

- $T^w_{\texttt{linSearch}}(n)$ ist in $\mathcal{O}(n)$, denn

  $T^w_{\texttt{linSearch}}(n) = 6n + 4$ und $c = 7$ und $n_0 = 3$ erfüllen die Definition:

  $$0 \leq T^w_{\texttt{linSearch}}(n) = 6n + 4 \leq 6n + n = \underbrace{7 \cdot n}_{=c \cdot n} \text{ für alle } n > \underbrace{3}_{=n_0}$$

- $S^w_{\texttt{linSearch}}(n)$ ist in $\mathcal{O}(1)$, denn

  $S^w_{\texttt{linSearch}}(n) = 2$ und $c = 2$ und $n_0 = 1$ erfüllen die Definition:

  $$0 \leq S^w_{\texttt{linSearch}}(n) = 2 \leq \underbrace{2 \cdot 1}_{=c \cdot 1} \text{ für alle } n > \underbrace{1}_{=n_0}$$

$\mathcal{O}$-Notation: Beispiele (2)

Definition $(\mathcal{O}(f(n)))$

Eine Funktion $g(n)$ ist in $\mathcal{O}(f(n))$, falls es Zahlen $c > 0$ und n_0 gibt, so dass $0 \leq g(n) \leq c \cdot f(n)$ für alle $n > n_0$.

Beispiele

- $g(n) = n^2 + 10n + 20$ ist in $\mathcal{O}(n^2)$, denn

  mit $c = 31$ und $n_0 = 1$:

  $$0 \leq n^2 + 10n + 20 \leq n^2 + 10n^2 + 20n^2 = \underbrace{31 \cdot n^2}_{=c \cdot n} \text{ für alle } n > \underbrace{1}_{=n_0}$$

- $g(n) = n^3$ ist **nicht** in $\mathcal{O}(n^2)$:

  Es gibt kein c und n_0, die die Definition erfüllen,
  denn aus $0 \leq n^3 \leq c \cdot n^2$ folgt $n \leq c$ und es gibt immer ein n, dass das verletzt!

Rechnen mit der $\mathcal{O}$-Notation

1. Wenn $g(n)$ in $O(f(n))$ ist, dann gilt für jede Konstante c $g(n) + c$ in $\mathcal{O}(f(n))$ und $c \cdot g(n)$ ist in $\mathcal{O}(f(n))$
2. Wenn $g_1(n)$ und $g_2(n)$ jeweils in $\mathcal{O}(f(n))$ sind, dann gilt auch: $g_1(n) + g_2(n)$ ist in $\mathcal{O}(f(n))$
3. Sei $g_1(n)$ in $\mathcal{O}(f_1(n))$ und $g_2(n) in \mathcal{O}(f_2(n))$. Dann gilt:
   - $h(n) = g_1(n) + g_2(n)$ ist in $\mathcal{O}(f_1(n) + f_2(n))$
   - $k(n) = g_1(n) \cdot g_2(n)$ ist in $\mathcal{O}(f_1(n) \cdot f_2(n))$
4. Sei $f(n)$ in $\mathcal{O}(g(n))$ und $g(n)$ in $\mathcal{O}(h(n))$.
   Dann ist $f(n)$ in $\mathcal{O}(h(n))$.
5. Sei $g(n) = p_k \cdot n^k + p_{k-1} \cdot n^{k-1} + \ldots + p_0$ ein Polynom von Grad k.
   Dann ist $g(n)$ in $\mathcal{O}(n^k)$

Komplexitätsklassen

Man nennt eine Funktion f

konstant	falls	$f(n) \in \mathcal{O}(1)$
logarithmisch	falls	$f(n) \in \mathcal{O}(\log n)$
linear	falls	$f(n) \in \mathcal{O}(n)$
log-linear	falls	$f(n) \in \mathcal{O}(n \log n)$
quadratisch	falls	$f(n) \in \mathcal{O}(n^2)$
kubisch	falls	$f(n) \in \mathcal{O}(n^3)$
polynomiell	falls	$f(n) \in \mathcal{O}(n^k)$ für ein k≥ 0
exponentiell	falls	$f(n) \in \mathcal{O}(k^n)$ für ein $k \geq 2$

Beachte:

$$\mathcal{O}(1) \subseteq \mathcal{O}(\log n) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^k) \subseteq \mathcal{O}(k^n) \text{ für alle } k \geq 2$$

Vergleich häufig auftretender Zeitkomplexitäten

$f(n)$	Bezeichnung	$f(10)$	$f(100)$	$f(1.000)$	$f(10.000)$
1	konstant	1sec	1sec	1sec	1sec
$\log_2 n$	logarithmisch	3sec	7sec	10sec	13sec
n	linear	10sec	100sec	17min	2,8hours
$n \log_2 n$	log-linear	33sec	11min	2,8hours	1,5 days
n^2	quadratisch	100sec	2,7hours	11,5days	3,2years
n^3	kubisch	17min	11,5days	31,7years	31.710years
2^n	exponentiell	17min	$>$ als Alter des Universums		

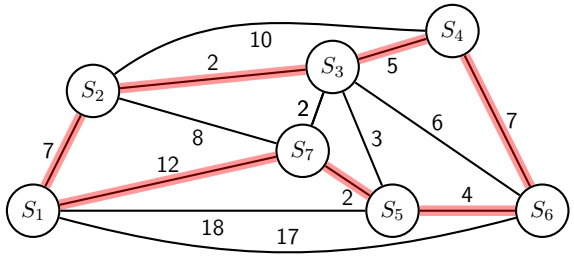
unter der Annahme: 1 Operation kostet 1sec

Exponentielle und polynomielle Komplexität

Problem des „Handelsreisenden“ (Traveling-Salesman-Problem)

Gegeben sei ein Graph mit n Städten und den jeweiligen Entfernungen zwischen den Städten.

Finde eine kürzeste Tour, so dass jede Stadt einmal besucht wird.



Für dieses Problem sind nur deterministische Algorithmen mit exponentieller Zeitkomplexität bekannt.

Bemerkungen zu TSP

- Für das Traveling-Salesman-Problem gibt es einen nichtdeterministischen polynomiellen (**NP**) Algorithmus („man darf die richtige Lösung raten“)
- Das Traveling-Salesman-Problem ist **NP-vollständig**: Falls es einen polynomiellen Algorithmus zu seiner Lösung gibt, so hat **jeder** nichtdeterministisch-polynomielle Algorithmus eine polynomielle Lösung.
- Die Frage, ob ein NP-vollständiges Problem (und damit alle) in polynomieller Zeit (**P**) gelöst werden kann, ist eine der bekanntesten ungelösten Fragen der theoretischen Informatik „**P** = **NP**“
- Das „**P** = **NP**“ ist eines der sieben sogenannten Millenium-Probleme

Binäre Suche in einem geordneten Array

Sei a ein **geordnetes** Array mit den Grenzen j und k, d.h. `a[i]` $<$ `a[i+1]` für $i = j, \ldots, k$; also z.B.:

a:	3	7	13	15	20	25	28	29
	j	$j+1$	$\cdots$					k

Algorithmus:
Um den Wert e in a zu suchen, teile das Array in der Mitte und vergleicht e mit dem Element `a[mid]` in der Mitte

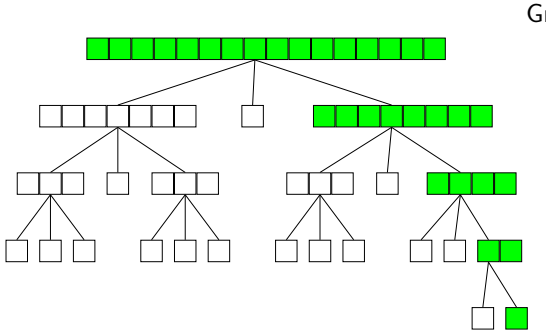
- Ist e $<$ `a[mid]`, dann suche weiter im linken Teil `a[j]...a[mid-1]`.
- Ist e $=$ `a[mid]`, dann hat man das Element gefunden.
- Ist e $>$ `a[mid]`, dann suche weiter im rechten Teil `a[mid+1]...a[k]`.

Binäre Suche in Java

```
static boolean binarySearch(int[] a, int e) {
 int j = 0;                    // linke Grenze
 int k = a.length - 1;         // rechte Grenze
 boolean found = false;        // wurde e schon gefunden?
 // solange e nicht gefunden und Array nicht leer
 while (!found && j <= k) {
  int mid = j + (k - j)/2;     // Mitte des Arrays bzw. links davon
  if (e < a[mid]) {            // e kleiner als das mittlere?
   k = mid - 1;                // ja, mache mit linkem Teil weiter.
  }
  else {
   if (e > a[mid]){            // e groesser als das mittlere?
    j = mid + 1;               // ja, mache mit rechtem Teil weiter.
   }
   else {                      // Sonst: gefunden
    found = true;
   }
  }
 return found;
}
```


Wie oft wird die while-Schleife maximal durchlaufen?

Beispiel: In einem Array der Größe 16 braucht man maximal 5 Durchläufe.



- Ein Array der Länge n mit $2^i \leq n < 2^{i+1}$ benötigt man im schlechtesten Fall $i+1$ Schleifendurchläufe
- Sei $\log_2(n)$ der ganzzahlige Anteil des Logarithmus zur Basis 2
- Es gilt $2^{\log_2(n)} \leq n < 2^{\log_2(n)+1}$
- Daher braucht man im schlechtesten Fall $\log_2(n) + 1$ Durchläufe.

Worst-Case-Komplexitäten der Suchalgorithmen

Laufzeitkomplexitäten

- $T^w_{\texttt{binarySearch}}(n)$ ist in $\mathcal{O}(\log(n))$ logarithmisch
- $T^w_{\texttt{linSearch}}(n)$ ist in $\mathcal{O}(n)$ linear

Speicherplatzkomplexitäten

- $S^w_{\texttt{binarySearch}}(n)$ ist in $\mathcal{O}(1)$ konstant
- $S^w_{\texttt{linSearch}}(n)$ ist in $\mathcal{O}(1)$ konstant

Beachte: Bei der Speicherplatzkomplexität zählen wir die Größe der Eingabe nicht mit.

Sortieren eines Arrays durch Einfügen (Insertion Sort)

Sei `int[] a` das zu sortierende Array.

Idee: Für alle i ab 1 bis zum Ende des Arrays:

- Seien die Elemente in `a[0], ..., a[i-1]` schon sortiert
- Nehme das Element a[i] an i-ter Position und füge es an der richtigen Stelle ein, sodass die Folge `a[0], ..., a[i]` sortiert ist.

Beachte: Für das Einfügen an der richtigen Stelle, muss Platz geschaffen werden:

Verschiebe die Elemente dafür nach rechts.

Insertion Sort: Beispiel

Am Anfang: Der erste Element ist immer schon sortiert:

5	33	12	13	8	1

Nächster Schritt: Füge 33 in den sortierten Bereich ein

5	33	12	13	8	1

Füge 12 in den sortierten Bereich ein: 33 muss verschoben werden

5	33		13	8	1
5		33	13	8	1
5	12	33	13	8	1

Füge 13 in den sortierten Bereich ein: 33 muss verschoben werden

5	12	33		8	1
5	12		33	8	1
5	12	13	33	8	1

Insertion Sort: Beispiel (2)

Füge 8 in den sortierten Bereich ein:
12,13,33 müssen verschoben werden

5	12	13	33		1
5	12	13		33	1
5	12		13	33	1
5		12	13	33	1
5	8	12	13	33	1

Insertion Sort: Beispiel (3)

Füge 1 in den sortierten Bereich ein:
5,8,12,13,33 müssen verschoben werden

5	8	12	13	33	
5	8	12	13		33
5	8	12		13	33
5	8		12	13	33
5		8	12	13	33
	5	8	12	13	33
1	5	8	12	13	33

Insertion Sort in Java

```
public static void insertionSort(int[] a) {
  for (int i = 1; i < a.length ; i++) {
    int toInsert = a[i]; // einzufuegendes Element
    int j = i;
    while (j > 0 && a[j-1] > toInsert) {
     a[j] = a[j-1]; // schiebe a[j-1] um 1 nach
         rechts
     j--;
    }
    a[j] = toInsert; // einfuege Position gefunden
    }
}
```


Komplexitäten des Insertion Sort

Sei n die Länge des Arrays

Zeitkomplexität:

Die äußere for-Schleife wird in jedem Fall $n-1$ mal durchlaufen.

Im i-ten Schritt wird die while-Schleife höchstens i mal durchlaufen, was durch n nach oben abschätzt werden kann.

Folglich ist die Zeitkomplexität des Insertion Sort in jedem Fall quadratisch, also in $\mathcal{O}(n^2)$.

Platzbedarf:

Neben der Eingabe werden 3 zusätzliche Speicherplätze für die lokalen Variablen gebraucht.
Daher ist die Speicherplatzkomplexität in der Eingabe n, konstant, d.h. in $\mathcal{O}(1)$.

Insertion Sort: Best- und Worst-Case

- Der beste Fall tritt ein, wenn die Eingabe bereits sortiert ist:
  Die while-Schleife wird in diesem Fall nie durchlaufen
  Die best-case Laufzeit von Insertion Sort ist in $\mathcal{O}(n)$
- Der schlechteste Fall tritt ein, wenn die Eingabe **absteigend** sortiert ist:
  Dann muss die while-Schleife jeweils i mal durchlaufen werden.
  Wie schon festgestellt, ist die worst-case Laufzeit in $\mathcal{O}(n^2)$

Sortieren eines Arrays durch Vertauschen (Bubble Sort)

Idee:
Vertausche benachbarte Elemente, wenn sie nicht wie gewünscht geordnet sind. In jedem Durchlauf des Feldes steigt das relativ größte Element wie eine Blase (bubble) im Wasser auf.

Algorithmus:

Sei „`outer`“ ein Zeiger auf das letzte Element des Arrays.

Solange „`outer`“ nicht auf das erste Element zeigt:

Sei „`inner`“ ein Zeiger auf das erste Element der Arrays.

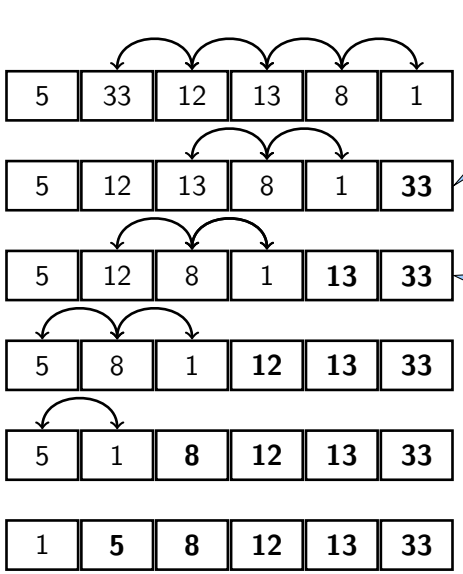
Solange „`inner` $<$ outer“:

Vertausche Elemente an den Positionen „`inner`“ und „`inner`$+1$“, wenn sie in falscher Reihenfolge stehen.

Rücke mit „`inner`“ eine Position vorwärts.

Rücke mit „outer“ eine Position rückwärts.

Bubble Sort: Beispiel



Nach dem ersten Durchlauf ist das größte Element an der richtige Stelle

Nach dem zweiten Durchlauf sind die beiden größten Elemente an der richtigen Stelle

In jedem Durchlauf werden höchstens n Vertauschungen durchgeführt

Bubble Sort in Java

```
static void bubbleSort(int[] a){
  for (int outer = a.length - 1; outer > 0; outer--) {
    for (int inner = 0; inner < outer; inner++) {
      if (a[inner] > a[inner + 1]) {
        // tausche a[inner] und a[inner + 1]
        int temp = a[inner];
        a[inner] = a[inner + 1];
        a[inner + 1] = temp;
      }
    }
  }
}
```


Komplexitäten des Bubble Sort

Sei n die Länge des Arrays

Zeitkomplexität:

Die äußere for-Schleife wird in jedem Fall $n+1$ mal durchlaufen.

Im i-ten Schritt wird die innere for-Schleife $n-i$ mal durchlaufen, was durch n nach oben abschätzt werden kann.

Folglich ist die Zeitkomplexität des Bubble Sort in jedem Fall quadratisch, also in $\mathcal{O}(n^2)$.

Beachte: Das ist auch die best-case Laufzeit von Bubble Sort!

Platzbedarf:

Neben der Eingabe werden 3 zusätzliche Speicherplätze für die lokalen Variablen gebraucht.

Daher ist die Speicherplatzkomplexität in der Eingabe n, konstant, d.h. in $\mathcal{O}(1)$.

Bubble Sort, optimiert

Wenn in einem Durchlauf keine Vertauschungen mehr passieren, dann ist das Feld bereits sortiert, und wir können aufhören:

```
static void bubbleSortOpt(int[] a){
  for (int outer = a.length - 1; outer > 0; outer--) {
    boolean notSwapped = true;
    for (int inner = 0; inner < outer; inner++) {
      if (a[inner] > a[inner + 1]) {
        // tausche a[inner] und a[inner + 1]
        notSwapped = false;
        int temp = a[inner];
        a[inner] = a[inner + 1];
        a[inner + 1] = temp;
      }
    }
    if notSwapped {return;}
  }
}
```

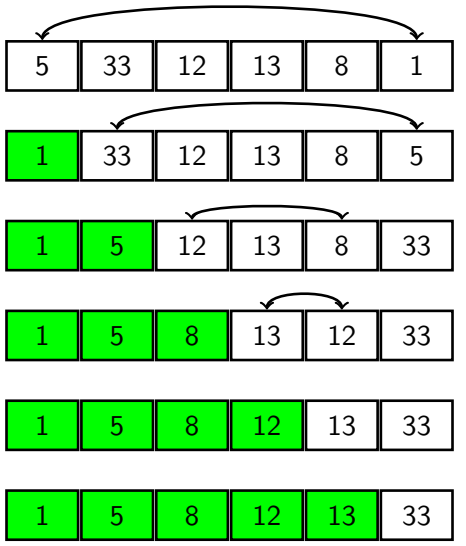
Beachte: Die Best-Case-Laufzeit verbessert sich dadurch von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n)$, die Wort-Case-Laufzeit bleibt bei $\mathcal{O}(n^2)$.

Sortieren durch Auswählen: Selection-Sort

Idee:

- Sortiere das Array von links nach rechts.
- Suche stets das minimale Element im unsortierten Teil des Arrays
- Das gefundene minimale Element wird gewählt und mit dem **ersten** Element des unsortierten Teils vertauscht.
- Die Länge des unsortierten Teils wird **um eins kürzer**.

Selection Sort: Beispiel



Selection Sort in Java

```
static void selectionSort(int[] a) {
  for (int i = 0; i < a.length - 1; i++) {
    int minIndex = selectMinIndex(a,i);
    int tmp = a[i];
     //vertauschen
     a[i] = a[minIndex];
     a[minIndex] = tmp;
  }
}
static int selectMinIndex(int[] a, int ab) {
  int minIndex = ab;
  for (int i = ab + 1; i < a.length; i++) {
    if (a[i] < a[minIndex]) {
       minIndex = i;
    }
  }
  return minIndex;
}
```


Komplexität des Selection Sort

Sei n die Länge des Arrays

Zeitkomplexität:

Es wird in jedem Fall n mal das Minimum gesucht und an die richtige Stelle getauscht.
Für jede Minimumsuche in der j-ten Runde wird ein Array der Größe $n - j$ durchsucht. Die Laufzeit hierfür kann mit n nach oben abgeschätzt werden.
Folglich ist die Zeitkomplexität des Selection Sort in jedem Fall quadratisch, also in $\mathcal{O}(n^2)$.
Beachte: Das ist auch die Best-Case Laufzeit!

Platzbedarf:

Neben der Eingabe werden nur konstant viele Speicherplätze benötigt. Daher ist die Speicherplatzkomplexität in der Eingabe n, konstant, d.h. in $\mathcal{O}(1)$.

Komplexitäten der drei Sortierverfahren

	Laufzeit		
	Worst-Case	Best-Case	Platz
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Bubble Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Bubble Sort mit Optimierung	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$

Zusammenfassung

- Lineare Suche in beliebigen Folgen
- Binäre Suche in geordneten Folgen
- Sortieren von Arrays:
  Insertion Sort, Bubble Sort und Selection Sort
- Komplexitätsanalyse und $\mathcal{O}$-Notation
- Laufzeit- und Speicherplatzkomplexität