Kapitel 10

Komplexität von Algorithmen und Sortieralgorithmen

Ziele

- Komplexität von Algorithmen bestimmen können
  (in Bezug auf Laufzeit und auf Speicherplatzbedarf)
- Sortieralgorithmen kennenlernen: Bubble Sort und Selection Sort
  (Quicksort wird in Kap. 11 behandelt)
- Komplexität von Sortieralgorithmen verstehen

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) Stack
  - Objekte (einschließlich Arrays) und deren Attribute, Heap
  - organisatorische Daten (davon abstrahieren wir im Folgenden).

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;
}
```

Zeitbedarf

Sei n = `a.length`
1 Zuweisung an `i`

2*n Arrayzugriffe +
2*n Vergleiche +
n Operationen (`i+1`) +
n Zuweisungen an `i` +
1 Arrayzugriff+1 Vergleich
wenn i = a.length
1 Return

Falls e nicht im Array:
1 + 6n + 2 + 1
= 6n + 4

Zeitbedarf für verschiedene Aufrufe der Methode `linSearch`:

```
int[] a = {30, 7, 1, 15, 20, 13, 28, 25};
boolean b1 = linSearch(a, 30);   // Zeit: 6
boolean b2 = linSearch(a, 23);   // Zeit: 52
```
= 6 * 8 + 4

Beispiel: Lineare Suche eines Elements (2)

	Speicherplatzbedarf
`static boolean linSearch(int[] a, int e){`	n+1 (für a mit `length`) 1 (für e)
`  for (int i = 0; i < a.length; i++){`	1 (für `i`)
`    if (a[i] == e){`	
`      return true;`	1 (für Ergebnis)
`    }`	
`  }`	$n+4$
`  return false;`	
`}`	

Speicherplatzbedarf für verschiedene Aufrufe der Methode `linSearch`:

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

$= 8 + 4$

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)? also nicht vom konkreten Wert e
- 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

worst case

average case

- Zeitkomplexität im

schlechtesten Fall: $T^{\mathsf{w}}_{\mathsf{m}}(n) = \max\{T_{\mathsf{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$

mittleren Fall: $T^{\mathsf{a}}_{\mathsf{m}}(n) = \text{Durchschnitt von } \{T_{\mathsf{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$

besten Fall: $T^{\mathsf{b}}_{\mathsf{m}}(n) = \min\{T_{\mathsf{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$

best case

- Speicherplatzkomplexität im

schlechtesten Fall: $S^{\mathsf{w}}_{\mathsf{m}}(n) = \max\{S_{\mathsf{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$

mittleren Fall: $S^{\mathsf{a}}_{\mathsf{m}}(n) = \text{Durchschnitt von } \{S_{\mathsf{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$

besten Fall: $S^{\mathsf{b}}_{\mathsf{m}}(n) = \min\{S_{\mathsf{m}}(e) \mid \text{Größe von } e \text{ ist } n\}$

Beispiel: Lineare Suche

```
static boolean linSearch(int[] a, int e)
```

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

<u>Speicherplatzbedarf:</u> $S^{w}_{\texttt{linSearch}}(n) = S^{a}_{\texttt{linSearch}}(n) = S^{b}_{\texttt{linSearch}}(n) = n+4$

<u>Zeitbedarf:</u>

Schlechtester Fall: $T^{w}_{\texttt{linSearch}}(n) = 2+(6*n+2) = 6*n+4$

Bester Fall: $T^{b}_{\texttt{linSearch}}(n) = 6$

für jede natürliche Zahl n

Beispiel: Lineare Suche (Durchschnittlicher Zeitbedarf)

Summe des Zeit-
bedarfs für jeden
Fall

Zeitbedarf:

Durchschnittlicher Fall: $T^{a}_{\texttt{linSearch}}(n) = \dfrac{\sum_{j=1}^{n}(6*j)}{n} = 3*n+3$

Erklärung:

Wir nehmen an, dass das Element im Array vorkommt.
Bei einer Eingabe der Länge n gibt es n Möglichkeiten:

1 Schleifendurchlauf wird benötigt um das Element zu finden,
2 Schleifendurchläufe werden benötigt,
...
n Schleifendurchläufe werden benötigt.

Da, nach Annahme
irgendwann das Element
gefunden wird und dann i++
nicht mehr aus-
geführt wird

Wir nehmen an, dass jeder dieser Fälle gleich wahrscheinlich ist.
Im Fall von j Schleifendurchläufen (1 ≤ j ≤ n) werden 2+6*j-2 Zeiteinheiten benötigt. Der durchschnittliche Fall ergibt sich dann aus dem arithmetischen Mittel der Summe des Zeitbedarfs aller n Fälle für j = 1,...,n.

Größenordnung der Komplexität: Die O-Notation

- Eine *exakte* Beschreibung des Zeit- und Speicherplatzbedarfs wird schnell zu kompliziert um praktikabel zu sein.
- Die Komplexität der Funktionen $T^{\mathsf{w}}(n)$, $T^{\mathsf{a}}(n)$, $T^{\mathsf{b}}(n)$, und $S^{\mathsf{w}}(n)$, $S^{\mathsf{a}}(n)$, $S^{\mathsf{b}}(n)$ wird häufig nur bis auf konstante Faktoren untersucht.

  d.h. bis auf konstante Faktoren

- Sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion. Wir definieren $O(f(n))$ als die Klasse aller Funktionen, die nicht wesentlich schneller wachsen als $f(n)$:

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

  z.B. $g(n) = 6n + 4$
  $f(n) = n$

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

  Dann gilt für alle $n > 3 =: n_0$

  $\underbrace{g(n)}_{6n+4} \leq \underbrace{7}_{c} * \underbrace{f(n)}_{7n}$

  Beispiele:

  $T^{\mathsf{w}}_{\texttt{linSearch}}(n)$ ist in $O(n)$

  $S^{\mathsf{w}}_{\texttt{linSearch}}(n)$ ist in $O(n)$

$T^{b}_{\text{linSearch}}(n) \in O(1)$

Komplexitätsklassen

Man nennt eine Funktion f

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

wobei $O(1) \subseteq O(\log(n)) \subseteq O(n) \subseteq O(n^2) \subseteq O(n^k) \subseteq O(k^n)$ für alle $k \geq 2$

$\cap|$ $\cup|$

$O(n \log(n))$

Annahme: Bei Größe $n = 1$ dauert die Berechnung 1 sec.

Vergleich häufig auftretender Zeitkomplexitäten

f(n)		f(10)	f(100)	f(1000)	f(10^4)
1	konstant	1s	1s	1s	1s
$\log_2(n)$	logarithm.	≈3s	7s	10s	13s
n	linear	10s	100s ≈ 1min	16min	2h
$n*\log_2(n)$	log-linear	30s	10min	12h	1d
n^2	quadratisch	1min	2h	11d	3 Jahre
n^3	kubisch	16min	11d	30 Jahre	30.000 Jahre
2^n	exponentiell	16min	2^{100} > als	Alter	des Univer-sums

Die Zeitangaben sind ungefähre Werte.

Exponentielle und polynomielle Komplexität

Für folgendes Problem des „**Handelsreisenden**“ (engl. Traveling Salesman) sind nur (determinisitische) Algorithmen mit exponentieller Zeitkomplexität bekannt:

Gegeben sei ein Graph mit n Städten und den jeweiligen Entfernungen zwischen den Städten. Berechnet werden soll eine kürzeste Tour, so dass jede Stadt einmal besucht wird?

Randbemerkung:

Für das Traveling-Salesman-Problem gibt es einen **nichtdeterministisch-polynomiellen** (**NP**) Algorithmus („man darf die richtige Lösung raten“).

Das Traveling-Salesman-Problem ist **NP-vollständig**, d.h. 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?**“

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,…,k;` also z.B.:

mid

a:	3	7	13	15	20	25	28	29
	j	j+1	…					k

Algorithmus:

Um den Wert `e` in `a` zu suchen, teilt man das Array in der Mitte und vergleicht `e` mit dem Element in der Mitte :

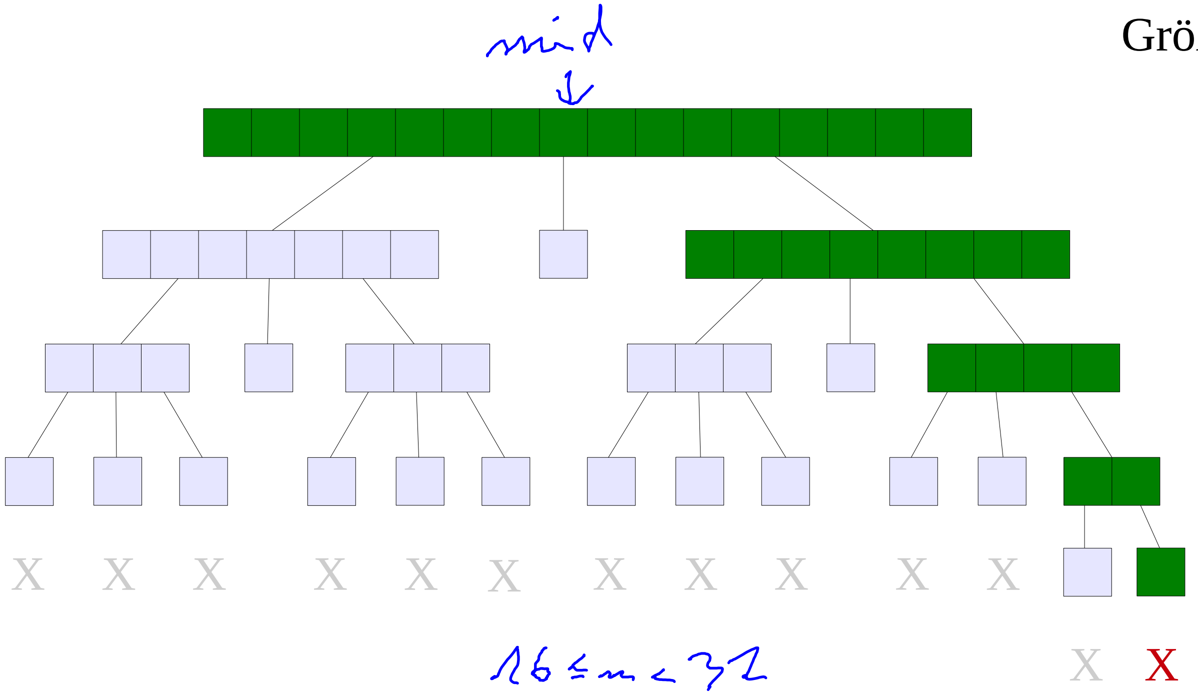
- Ist `e < a[mid],` so sucht man weiter im linken Teil `a[j],…,a[mid-1].`
- Ist `e = a[mid],` so hat man das Element gefunden.
- Ist `e > a[mid],` so sucht man weiter im rechten Teil `a[mid+1],…, a[k].`

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 das Element e schon gefunden?
  while (!found && j <= k) {  // solange nicht gefunden und Array nicht leer
     int mid = j + (k - j)/2; // Mitte des Arrays bzw. links von der Mitte
     if (e < a[mid]) {       // Ist e kleiner als das mittlere Element,
        k = mid - 1;        // so machen wir mit dem linken Teilarray weiter.
     } else if (e > a[mid]){ // Ist e groesser das mittlere Element, so
        j = mid + 1;          // machen wir mit dem rechten Teilarray weiter.
     } else {              //Anderenfalls haben wir den Wert im Array gefunden.
        found = true;
     }
  } // Ende while
  return found;
}
```

Wie oft wird die while-Schleife maximal durchlaufen?

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



Größe des betrachteten Arrays

16 bis 31	2^4 = $\log_2(16)$
	" $\log_2(31)$
8 bis 15	2^3
4 bis 7	2^2
2 bis 3	2^1
1 bis 1	2^0
0	

- Array der Länge n mit $2^i \leq n < 2^{i+1}$ benötigt im schlimmsten Fall i+1 Schleifendurchläufe.
- $\log_2(n)$ bezeichnet den ganzzahligen Anteil des Logarithmus zur Basis 2.
  Es gilt: $2^{\log_2(n)} \leq n < 2^{\log_2(n)+1}$
- Daher benötigt ein Array der Länge n im schlimmsten Fall $\log_2(n)$ + 1 Durchläufe.

Worst-Case Komplexitäten der Suchalgorithmen

$T^{w}_{\texttt{binarySearch}}(n)$ ist in O(log(n)) logarithmisch

$S^{w}_{\texttt{binarySearch}}(n)$ ist in O(n) linear

funktioniert nur für geordnete Array

$T^{w}_{\texttt{linSearch}}(n)$ ist in O(n) linear

$S^{w}_{\texttt{linSearch}}(n)$ ist in O(n) linear

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.

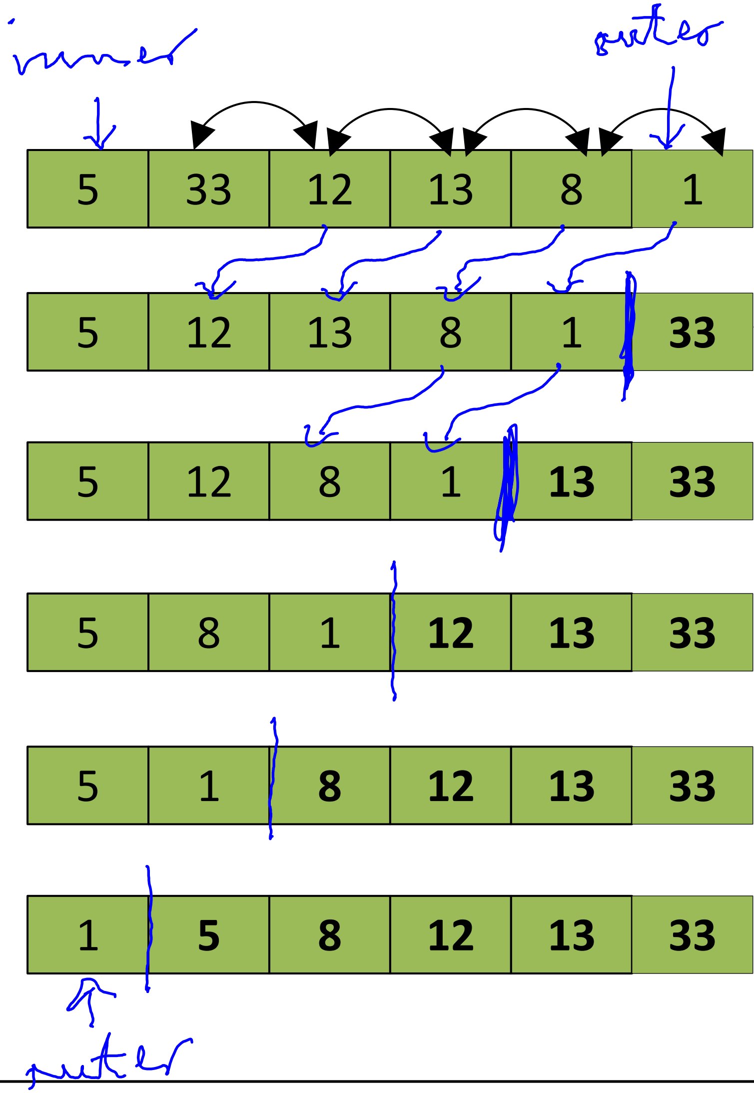
Algorithmus:

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

Solange „outer“ nicht auf das erste Element zeigt:

1. Sei „inner“ ein Zeiger auf das erste Element des Arrays.
   Solange „inner < outer“:
   1.1. Vertausche Elemente an den Positionen „inner“ und „inner+1“, wenn sie in falscher Reihenfolge stehen.
   1.1. Rücke mit „inner“ eine Position vorwärts.
2. Rücke mit „outer“ eine Position rückwärts.

Bubble Sort: Beispiel



Nach dem ersten Durchlauf ist das größte Element an der richtigen Stelle.

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

In jedem Durchlauf werden höchstens n Vertauschungen ausgeführt.

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;
         } //Ende if
      } // Ende innere for-Schleife
   } //Ende äußere for-Schleife
}
```

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 abgeschätzt werden kann.

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

Speicherplatzkomplexität:

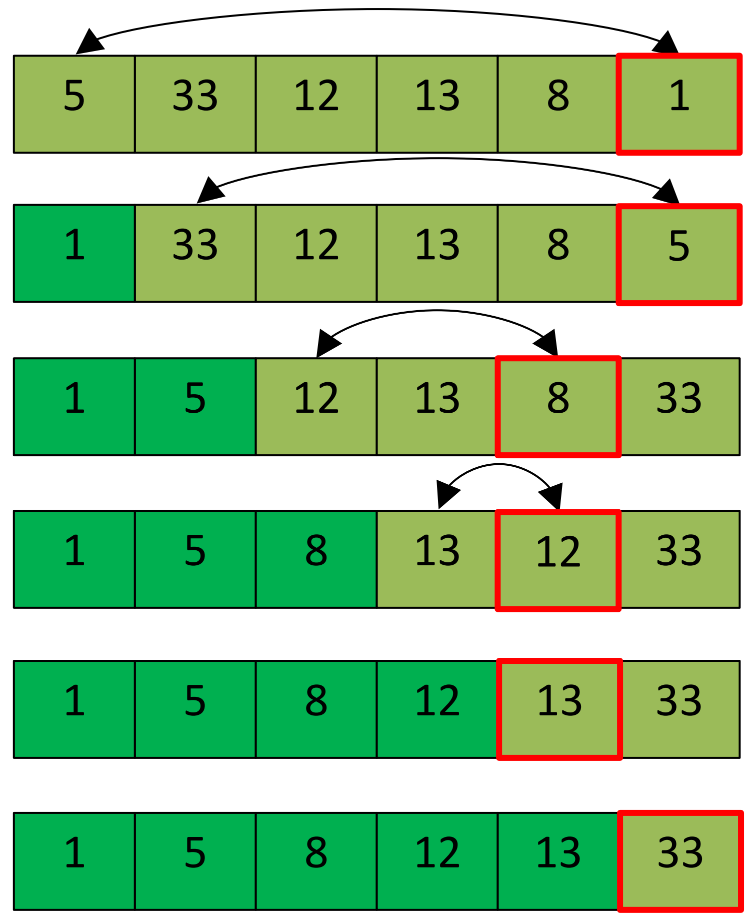
Es werden in jedem Fall n Speicherplätze für den Array und 3 zusätzliche Speicherplätze für die lokalen Variablen gebraucht.

Folglich ist die Speicherplatzkomplexität des Bubble Sort linear, d.h. in $O(n)$.

Sortieren eines Arrays durch Auswahl (Selection Sort)

Sortiere Array von links nach rechts.

- In jedem Schritt wird der noch **unsortierte** Teil des Arrays nach einem **minimalen** Element durchsucht.
- 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 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;}
```

Komplexitäten des Selection Sort

Sei n die Länge des Arrays.

<u>Zeitkomplexität:</u>

Es werden in jedem Fall n Schritte durchgeführt, wobei in jedem Schritt eine Minimumsuche in einem Teil des Arrays erfolgt.

Für die Minimumsuche im j-ten Schritt werden n-j Schritte durchgeführt, was durch n nach oben abgeschätzt werden kann.

Folglich ist die Zeitkomplexität des Selection Sort in jdem Fall quadratisch, also in $O(n^2)$.

<u>Speicherplatzkomplexität:</u>

Es werden in jedem Fall n Speicherplätze für den Array und eine konstante Zahl von zusätzlichen Speicherplätzen für die lokalen Variablen und Rückgaben gebraucht.

Folglich ist die Speicherplatzkomplexität des Selection Sort linear, d.h. in $O(n)$.