

Kapitel 9

Komplexität von Algorithmen und Sortieralgorithmen

Ziele

- Komplexität von Algorithmen bestimmen können
- Sortieralgorithmen kennenlernen: Bubble Sort und Selection Sort (Quicksort wird in Kap. 10 behandelt)

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

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

Zeitbedarf

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

Sei $n = a.length$

1 Zuweisung an i

\leq (2*n+1 Vergleiche +
2*n+1 Arrayzugriffe +
n Operationen ($i+1$) +
n Zuweisungen an i)

1 Return

Zeitbedarf für verschiedene Aufrufe der Methode `linSearch`:

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

Beispiel: Lineare Suche eines Elements (2)

	<u>Speicherplatzbedarf</u>
<pre>static boolean linSearch(int[] a, int e){</pre>	n+1 (für a und e)
<pre> for (int i = 0; i < a.length; i++){</pre>	}
<pre> if (a[i] == e){</pre>	
<pre> return true;</pre>	
<pre> }</pre>	1 (für i)
<pre> }</pre>	
<pre> return false;</pre>	
<pre>}</pre>	1 (für Ergebnis)

Speicherplatzbedarf für verschiedene Aufrufe der Methode `linSearch`:

```
int[] a = new int[] {30, 7, 1, 15, 20, 13, 28, 25};  
boolean b1 = linSearch(a, 30); // Speicherplätze: 11  
boolean b2 = linSearch(a, 23); // Speicherplätze: 11
```

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: $T_m(e)$ — Zeitbedarf des Methodenaufrufs $m(e)$
 $S_m(e)$ — Speicherplatzbedarf des Methodenaufrufs $m(e)$
- 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: $T_m^w(n) = \max \{T_m(e) \mid \text{Größe von } e \text{ ist } n\}$

mittleren Fall: $T_m^a(n) = \text{Durchschnitt von } \{T_m(e) \mid \text{Größe von } e \text{ ist } n\}$

besten Fall: $T_m^b(n) = \min \{T_m(e) \mid \text{Größe von } e \text{ ist } n\}$

■ Speicherplatzkomplexität im

schlechtesten Fall: $S_m^w(n) = \max \{S_m(e) \mid \text{Größe von } e \text{ ist } n\}$

mittleren Fall: $S_m^a(n) = \text{Durchschnitt von } \{S_m(e) \mid \text{Größe von } e \text{ ist } n\}$

besten Fall: $S_m^b(n) = \min \{S_m(e) \mid \text{Größe von } e \text{ ist } n\}$

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.)

Speicherplatzbedarf: $S_{\text{linSearch}}^w(n) = S_{\text{linSearch}}^a(n) = S_{\text{linSearch}}^b(n) = 11$

Zeitbedarf:

Schlechtester Fall: $T_{\text{linSearch}}^w(n) = 2 + (6 * n + 2) = 6 * n + 4$

Bester Fall: $T_{\text{linSearch}}^b(n) = 6$

Beispiel: Lineare Suche (Durchschnittlicher Zeitbedarf)

Zeitbedarf:

$$\text{Durchschnittlicher Fall: } T_{\text{linSearch}}^a(n) = 2 + \frac{\sum_{j=1}^n (6 * j + 2)}{n} = 3 * n + 7$$

Erklärung:

Bei einer Eingabe der Länge n gibt es n Fälle:

- 1 Schleifendurchlauf wird benötigt,
- 2 Schleifendurchläufe werden benötigt,
- ...
- n Schleifendurchläufe werden benötigt.

Wir nehmen an, dass jeder dieser Fälle gleich wahrscheinlich ist.

Im Fall von j Schleifendurchläufen ($1 \leq j \leq n$) werden $6*j+2$ Zeiteinheiten benötigt.

Der durchschnittliche Fall ergibt sich dann aus dem arithmetischen Mittel der Summe des Zeitbedarfs aller n Fälle für $j = 1, \dots, n$.

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 $T^w(n)$, $T^a(n)$, $T^b(n)$, und $S^w(n)$, $S^a(n)$, $S^b(n)$ wird häufig nur bis auf konstante Faktoren untersucht.
- Sei $f: \mathbb{N} \rightarrow \mathbb{N}$ 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 gibt, so dass $0 \leq g(n) \leq c \cdot f(n)$ für alle $n > n_0$ gilt.

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

Beispiele:

$$T_{\text{linSearch}}^w(n) \text{ ist in } O(n)$$

$$S_{\text{linSearch}}^w(n) \text{ ist in } O(n)$$

Komplexitätsklassen

Man nennt eine Funktion f

konstant	falls	$f(n) \in O(1)$
logarithmisch	falls	$f(n) \in O(\log(n))$
linear	falls	$f(n) \in O(n)$
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 $k \geq 2$

Binäre Suche in einem geordneten Array

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

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 :

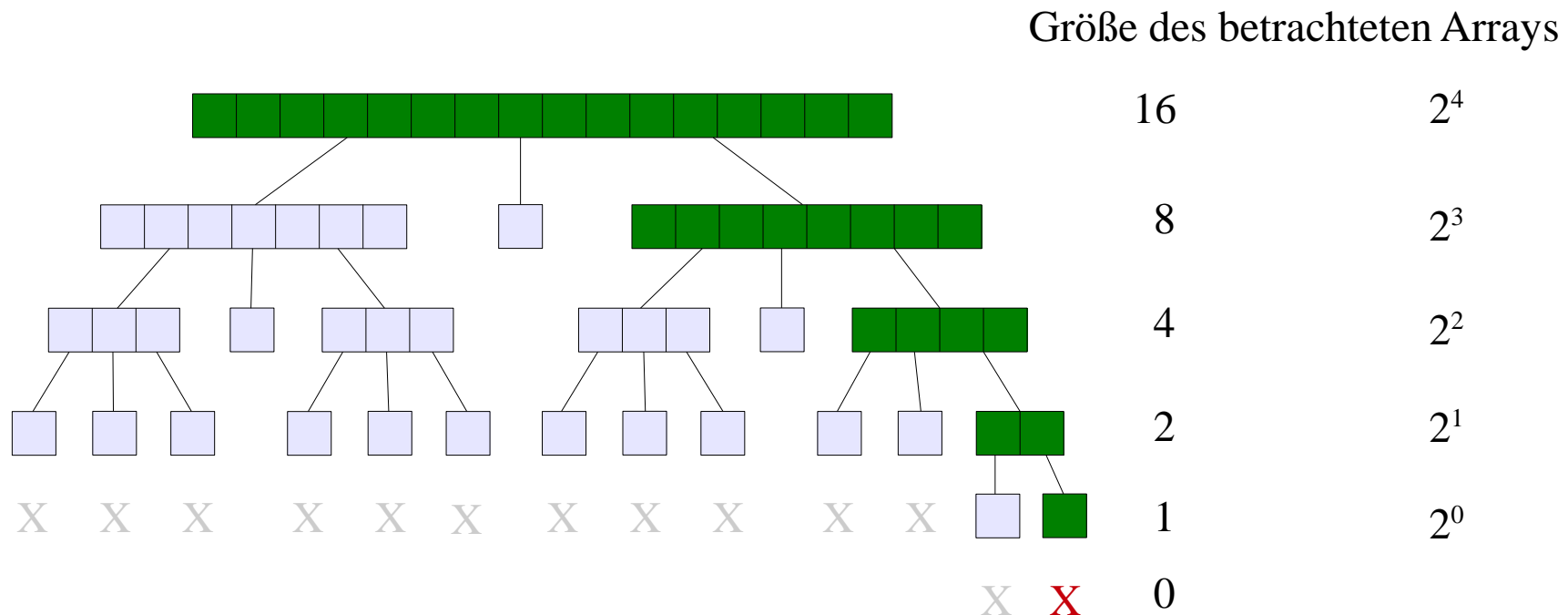
- Ist $e < a[\text{mid}]$, so sucht man weiter im linken Teil $a[j], \dots, a[\text{mid}-1]$.
- Ist $e = a[\text{mid}]$, so hat man das Element gefunden.
- Ist $e > a[\text{mid}]$, so sucht man weiter im rechten Teil $a[\text{mid}+1], \dots, a[k]$.

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.



- Array der Länge n mit $2^i \leq n < 2^{i+1}$ benötigt im schlimmsten Fall $i+1$ Schleifendurchläufe.
- Da $2^{\log_2(n)} \leq n < 2^{\log_2(n)+1}$ benötigt ein Array der Länge n im schlimmsten Fall den ganzzahligen Anteil von $\log_2(n) + 1$ Durchläufe.

Komplexitäten der Suchalgorithmen

$T_{\text{binarySearch}}^w(n)$ ist in $O(\log(n))$ logarithmisch

$S_{\text{binarySearch}}^w(n)$ ist in $O(n)$ linear

$T_{\text{linSearch}}^w(n)$ ist in $O(n)$ linear

$S_{\text{linSearch}}^w(n)$ ist in $O(n)$ linear

Vergleich häufig auftretender Komplexitäten

$f(n)$		$f(10)$	$f(100)$	$f(1000)$	$f(10^4)$
1	konstant	1	1	1	1
$\log_2(n)$	logarithm.	3	7	10	13
n	linear	10	100	1000	10^4
$n \cdot \log_2(n)$	log-linear	30	700	10^4	10^5
n^2	quadratisch	100	10^4	10^6	10^8
n^3	kubisch	1000	10^6	10^9	10^{12}
2^n	exponentiell	1000	10^{30}	10^{300}	10^{3000}

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	1min	16min	2h
$n \cdot \log_2(n)$	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	> als	Alter	des Univer- sums

Die Zeitangaben sind ungefähre Werte

Exponentielle und polynomielle Komplexität

Für folgendes Problem des „**Handelsreisenden**“ (engl. Traveling Salesman) sind nur Algorithmen mit exponentieller Zeitkomplexität bekannt:
Gegeben sei ein Graph mit n Städten und den jeweiligen Entfernungen sowie eine Entfernung B . Gibt es eine Tour der Länge $\leq B$ durch alle Städte, 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?**“

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.

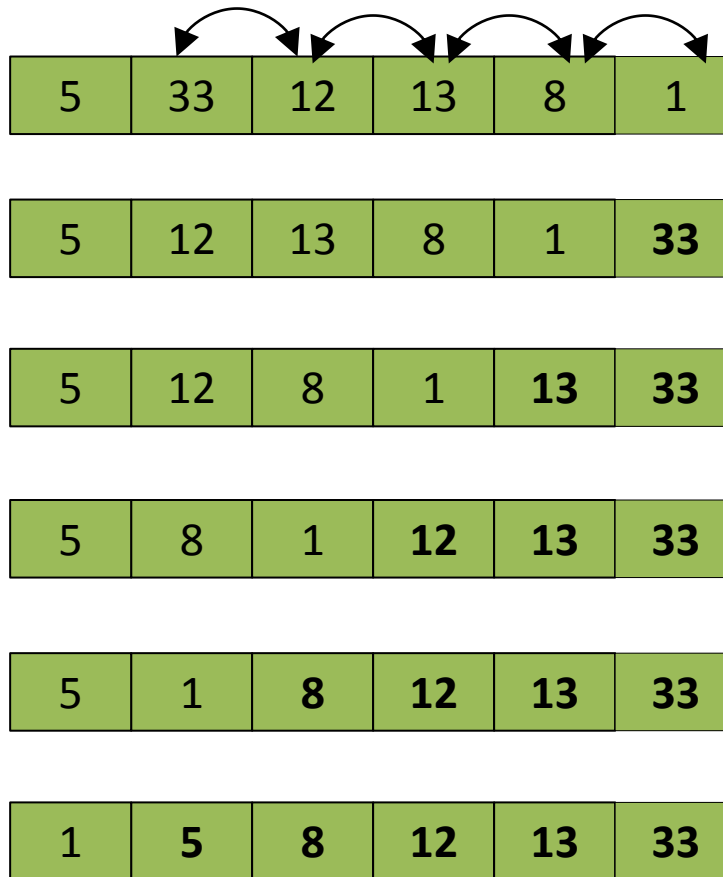
Solange „inner < outer“:

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.

Bubble Sort in Java

```
static void bubbleSort(double[] 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:

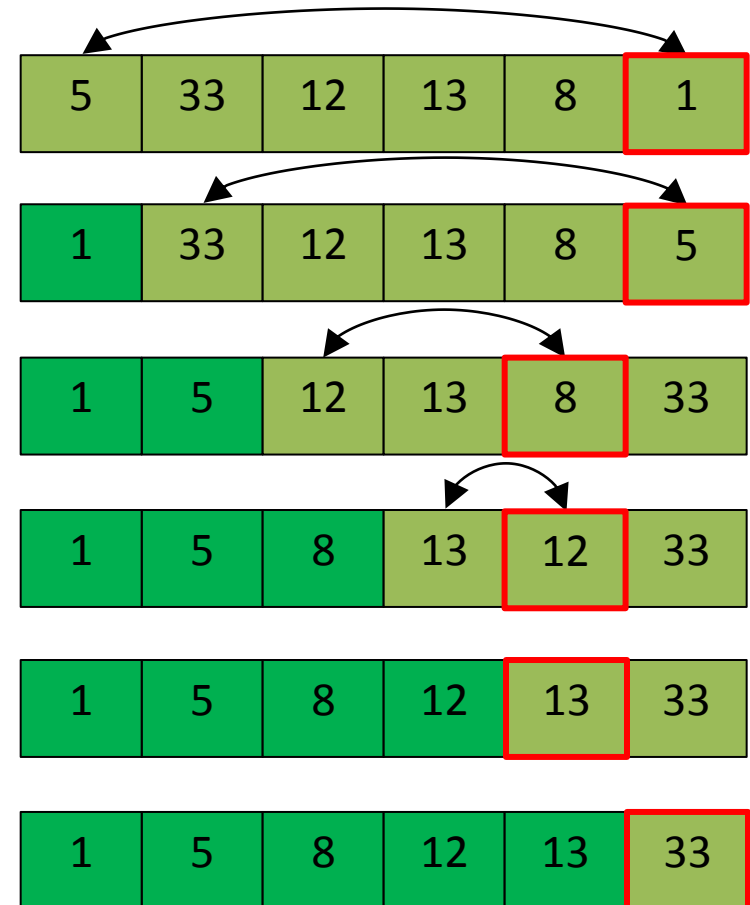
Es werden in jedem Fall n Speicherplätze für den Array und eine feste Zahl von zusätzlichen Speicherplätzen für die lokalen Variablen gebraucht.

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 dem **kleinsten** Element durchsucht.
- Das kleinste 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(double[] 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, 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.

Zeitkomplexität:

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 jedem Fall quadratisch, also in $O(n^2)$.

Speicherplatzkomplexität:

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 $O(n)$.