# LB 4 – Algorithmen in Lazarus

Word-Datei: Modul 0 - Sortieren (Grundidee und Visuelle Programme)\Sortierverfahren1.doc

Ein **Sortierverfahren** ist ein Algorithmus, der dazu dient, eine Liste von Elementen zu sortieren. Diese Liste kann aus Zahlen oder Zeichenketten bestehen.

**Beispiele aus der Praxis:**

- Artikel in einem Lager sortieren
- Bücherei → Autoren … (allg. Datensammlungen)
- Wörterbücher, Telefonbücher
- Statistik
- Suchmaschinen

→ Existenz verschiedene Sortierverfahren, diese arbeiten unterschiedlich effizient

**Laufzeit T(n)**

Algorithmen, bei denen sehr viele Operationen (z.B. Vergleiche und Vertauschungen) auszuführen sind, benötigen eine bestimmte Rechenzeit. Diese lässt sich relativ einfach über die Systemzeit des Computers bestimmen. Nachteil der Messung ist die Abhängigkeit von der Prozessorgeschwindigkeit, von der Zugriffszeit auf den Speicher und anderen hardwarerelevanten Gegebenheiten.

Um dennoch eine Vergleichbarkeit von Algorithmen zu ermöglichen, nutzt man die Ermittlung der Operationsanzahl, die selbstverständlich von der Anzahl der zu sortierenden Elemente **n** abhängt. Man spricht in diesem Zusammenhang dann von der ***Komplexität*** eines Algorithmus.

**Komplexität O(n)**
→ Komplexität eines Algorithmus (=die Anzahl der nötigen Operationen)

→ wird üblicherweise in der **Landau-Notation [ z.B. O(n) ]** dargestellt

→ einige Sortierverfahren benötigen für die Durchführung weiteren Speicherplatz.

→ Komplexität und Speicherbedarf hängen bei einigen Sortierverfahren von der anfänglichen Anordnung der Werte im Array ab, man unterscheidet:

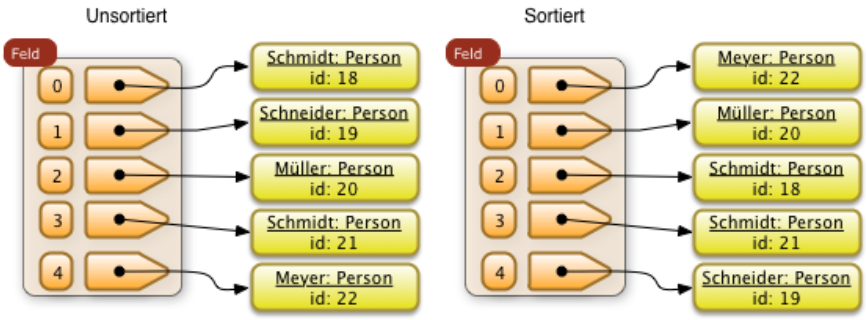
- ***Best Case*** (bester Fall) → bereits sortierte Liste vorhanden
- ***Average Case*** (Durchschnittsverhalten)
- ***Worst Case*** (schlechtester Fall) → eine umgekehrt sortierte Liste vorhanden (klei-größ→größ-klei).

→ Unterschied zwischen stabilen und instabilen Sortierverfahren:

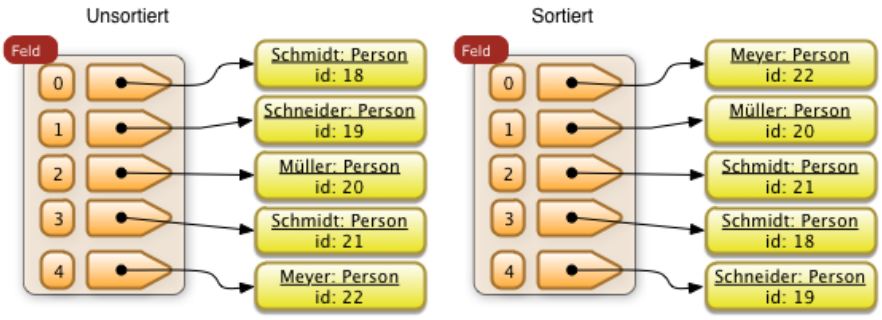
- ein Sortieralgorithmus ist stabil, wenn beim Sortieren gleichwertige Schlüsselelemente (Element a ist gleich Element b) niemals gegeneinander vertauscht werden
- instabile Sortierverfahren garantieren dies nicht

**Beispiel**: Eine Folge von Personen ist ursprünglich nach der Mitarbeiternummer (id) sortiert. Diese Folge soll mit dem Nachnamen als Sortierschlüssel neu sortiert werden.

Im folgenden Beispiel wurde ein stabiles Sortierverfahren angewendet. Die relative Ordnung der beiden Personen mit dem Namen "Schmidt" bleibt erhalten. Der Mitarbeiter mit der Nummer 18 bleibt vor dem Mitarbeiter mit der Nummer 21.




Bei nicht stabilen Sortierverfahren bleibt diese relative Ordnung nicht unbedingt erhalten. Hier könnte eine sortierte Folge so aussehen:



## 2. Der Sortieralgorithmus Min-Sort / Max-Sort (Selection-Sort)

Word-Datei: Modul 1 - Minsort\Minisort.doc

**Aufgabe 1**

Formuliere den *Algorithmus Minsort* verbal.

- Durchlaufe alle Werte von links nach rechts, kleinster Wert (Minimum) wird gemerkt
- Gemerkter Wert wird mit dem ersten Wert getauscht
- Alle Werte vom 2. bis letztem Wert werden auf den kleinsten Wert (Min.) geprüft
- Das Minimum wird mit dem zweiten Wert getauscht
- Minimum wird mit dem 2. Wert getauscht
- Wiederholung, bis alle Werte sortiert ist

# I. Vorstufe: Minimum suchen und nach vorne tauschen

| a | *1* | *2* | *3* | *4* | *5* | *6* | *7* | |
|---|---|---|---|---|---|---|---|---|
| | 33 | 27 | 39 | 19 | 22 | 13 | 21 | *unsortierte Zahlenfolge* |

### a) Minimum suchen

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *i := 1* | **33** | 27 | 39 | 19 | 22 | 13 | 21 | MinPosition := 1 |
| *i := 2* | 33 | **27** | 39 | 19 | 22 | 13 | 21 | MinPosition := 2 |
| *i := 3* | 33 | 27 | **39** | 19 | 22 | 13 | 21 | MinPosition := 2 |
| *i := 4* | 33 | 27 | 39 | **19** | 22 | 13 | 21 | MinPosition := 4 |
| *i := 5* | 33 | 27 | 39 | 19 | **22** | 13 | 21 | MinPosition := 4 |
| *i := 6* | 33 | 27 | 39 | 19 | 22 | **13** | 21 | MinPosition := 6 |
| *i := 7* | 33 | 27 | 39 | 19 | 22 | 13 | **21** | MinPosition := 6 |

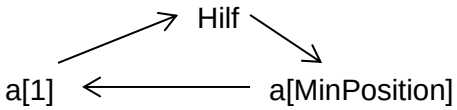
**Aufgabe 2 (Übung)**

Führe den Algorithmus für die Zahlenfolge

| 35 | 18 | 34 | 17 | 12 | 10 | 16 |
|---|---|---|---|---|---|---|

im Heft durch.

**b) Minimum mit dem ersten Element vertauschen**




**hilf:=a[1]**

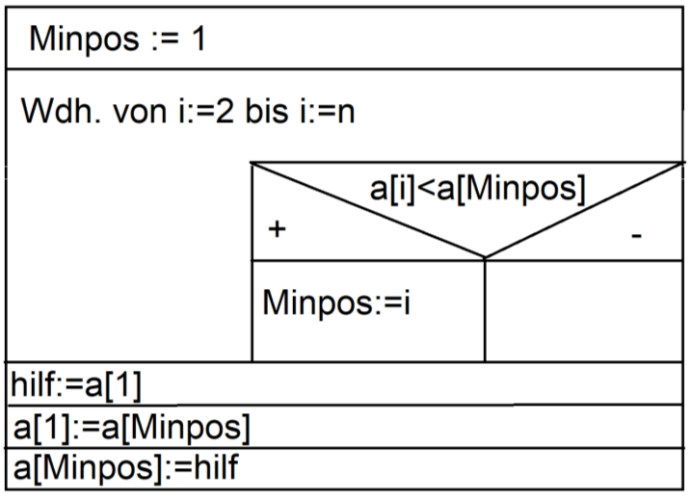
**a[1]:=a[Minpos]**

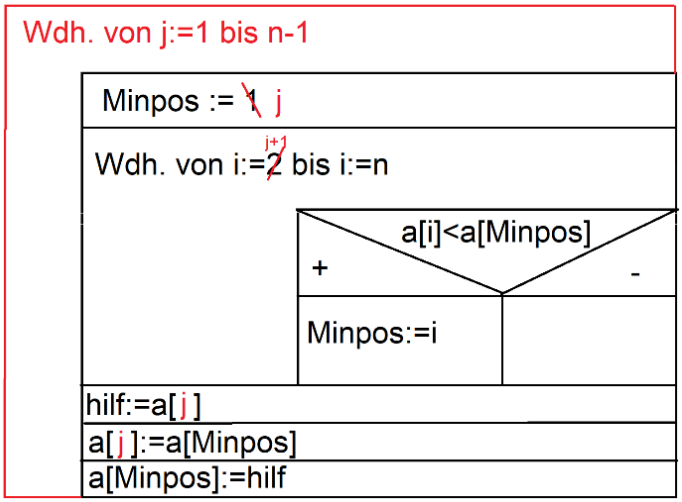
**a[Minpos] := hilf**

**Aufgabe 3**

Entwickle einen Algorithmus als Struktogramm, der das **Minimum** in einem Feld findet und an den Feldanfang tauscht.

**Minsort**



## II. Minsort (Sortieren durch Auswahl)

| a | *1* | *2* | *3* | *4* | *5* | *6* | *7* | |
|---|---|---|---|---|---|---|---|---|
| | 27 | 8 | 39 | 42 | 6 | 19 | 13 | *unsortierte Zahlenfolge* |

index := 1

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *erster Schritt:* | **6** | **8** | 39 | 42 | 27 | 19 | 13 | Minimum finden → Minpos = 2 |
| *zweiter Schritt:* | **6** | **8** | 39 | 42 | 27 | 19 | 13 | *Umspeichern mit index = 2* |

index := 3

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *erster Schritt:* | **6** | **8** | **39** | 42 | 27 | 19 | 13 | Minimum finden → Minpos = 7 |
| *zweiter Schritt:* | **6** | **8** | **13** | 42 | 27 | 19 | 39 | *Umspeichern mit index = 3* |

index := 4

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *erster Schritt:* | **6** | **8** | **13** | 42 | 27 | **19** | 39 | Minimum finden → Minpos = 6 |
| *zweiter Schritt:* | **6** | **8** | **13** | **19** | 27 | 42 | 39 | *Umspeichern mit index = 4* |

index := 5

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *erster Schritt:* | **6** | **8** | **13** | **19** | **27** | 42 | 39 | Minimum finden → Minpos = 5 |
| *zweiter Schritt:* | **6** | **8** | **13** | **19** | **27** | 42 | 39 | *Umspeichern mit index = 5* |

index := 6

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| *erster Schritt:* | **6** | **8** | **13** | **19** | **27** | 42 | **39** | Minimum finden → Minpos = 7 |
| *zweiter Schritt:* | **6** | **8** | **13** | **19** | **27** | **39** | 42 | *Umspeichern mit index = 6* |

**Aufgabe 4**

Führe den Algorithmus für die Zahlenfolge

| | | | | | | |
|---|---|---|---|---|---|---|
| 35 | 8 | 34 | 2 | 10 | 7 | 6 |

im Heft durch.

### Aufgabe 5

Entwickle ein Programm, das
**a)** ein Feld von Integer-Zahlen einliest,
**b)** mit dem Algorithmus *Minsort* sortiert und
**c)** sortiert wieder ausgibt.

### Aufgabe 6

Verändere die Prozedur *Minsort* so, dass eine Zahlenfolge in der Reihenfolge von der größten bis zur kleinsten Zahl sortiert wird (*Maxsort*).

### Aufgabe 7

Funktioniert der Algorithmus *Minsort* auch noch, wenn mehrere gleiche kleinste Zahlen im Feld vorhanden sind?

### Aufgabe 8

In einem Feld mit mehreren Minima sollen *alle* Positionen der Minima und deren Wert ausgegeben werden.

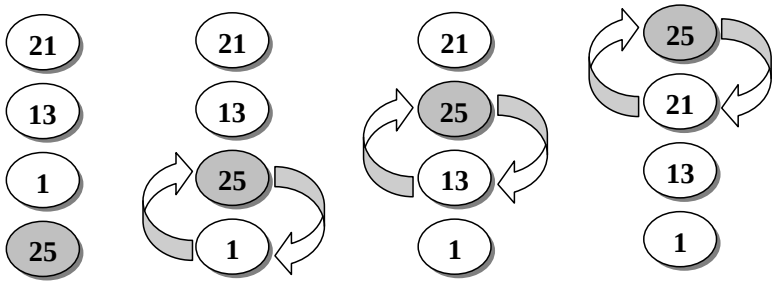
Entwickle ein Programm, das dies leistet.

## 3. Der Sortieralgorithmus "Bubblesort"

Word-Datei: ..\LB4\Modul 2 - Bubblesort\Bubblesort.doc

**Aufgabe 1**

Formuliere das Prinzip des Sortierens im abgebildeten Schema.




= Sortieren durch Nachbarschaftsverlgeich
Die größte Zahl wird erkannt und geht nun immer weiter und vergleicht jeden Wert. Wenn größer wird getauscht, wenn kleiner, dann bleibt die größte Zahl und wird mit der nächsten Zahl verglichen.

**Aufgabe 2**

Nimm Skat- oder nummerierte Karten oder das „Waageprogramm“ und sortiere eine Reihe nach der Idee des „Bubblesort“.

**Aufgabe 3**

Führe das begonnene Bubble-Sortieren des Beispielfeldes fort:

**erster Durchlauf:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 21 | 9 | 8 | 15 | 17 | 1 | |
| 5 | **21** | | | | | | links nicht größer → nicht tauschen |
| | 9 | **21** | | | | | links größer → tauschen |
| | | 8 | **21** | | | | links größer → tauschen |
| | | | 15 | **21** | | | links größer → tauschen |
| | | | | 17 | **21** | | links größer → tauschen |
| | | | | | 1 | **21** | links größer → tauschen |

Anzahl der Schleifendurchläufe der *inneren Schleife*: 6

**zweiter Durchlauf:**

| 5 | 9 | 8 | 15 | 17 | 1 | **21** | |
|---|---|---|---|---|---|---|---|
| 5 | **9** | | | | | | links nicht größer → nicht tauschen |
| | 8 | **9** | | | | | links größer → tauschen |
| | | 9 | **15** | | | | links nicht größer → nicht tauschen |
| | | | 15 | **17** | | | links nicht größer → nicht tauschen |
| | | | | 1 | **17** | 21 | links größer → tauschen |

Anzahl der Schleifendurchläufe der *inneren Schleife*: 5

**dritter Durchlauf:**

| 5 | 8 | 9 | 15 | 1 | **17** | **21** | |
|---|---|---|---|---|---|---|---|
| 5 | **8** | | | | | | Nicht tauschen |
| | 8 | **9** | | | | | Nt |
| | | 9 | **15** | | | | Nt |
| | | | 1 | **15** | | | Tauschen |

Anzahl der Schleifendurchläufe der *inneren Schleife*: 4

**vierter Durchlauf:**

| 5 | 8 | 9 | 1 | 15 | **17** | **21** | |
|---|---|---|---|---|---|---|---|
| 5 | **8** | | | | | | Nt |
| | 8 | **9** | | | | | Nt |
| | | 1 | **9** | | | | T |

Anzahl der Schleifendurchläufe der *inneren Schleife*: 3

**fünter Durchlauf:**

| 5 | 8 | 1 | 9 | 15 | **17** | **21** | |
|---|---|---|---|---|---|---|---|
| 5 | **8** | | | | | | Nt |
| | 1 | **8** | | | | | T |

Anzahl der Schleifendurchläufe der *inneren Schleife*: 2

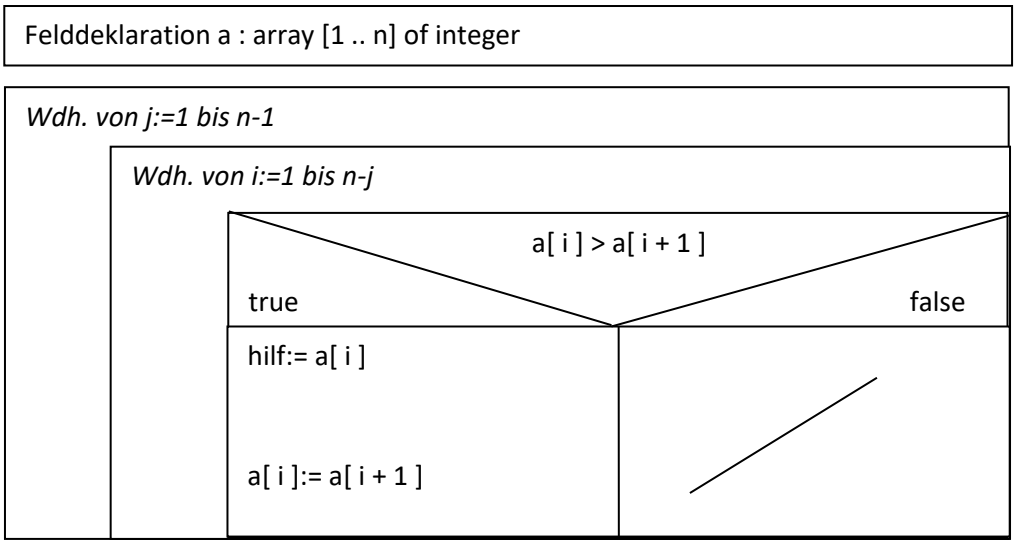
**sechster Durchlauf:**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | 1 | 8 | 9 | 15 | **17** | **21** | |
| 1 | **5** | **8** | 9 | 15 | 17 | 21 | T |

Anzahl der Schleifendurchläufe der *inneren Schleife*: 1

**Aufgabe 4**

Ergänze das ***Struktogramm*** für den Algorithmus "Bubblesort".




**Aufgabe 5**

Ergänzen Sie das Projekt ***Sortieren*** durch den Bubble-Sort-Algorithmus!

## Bubble-Sort, Selection-Sort, Insertion-Sort

→ P13_Sortieren_VL (pSort.lpi)

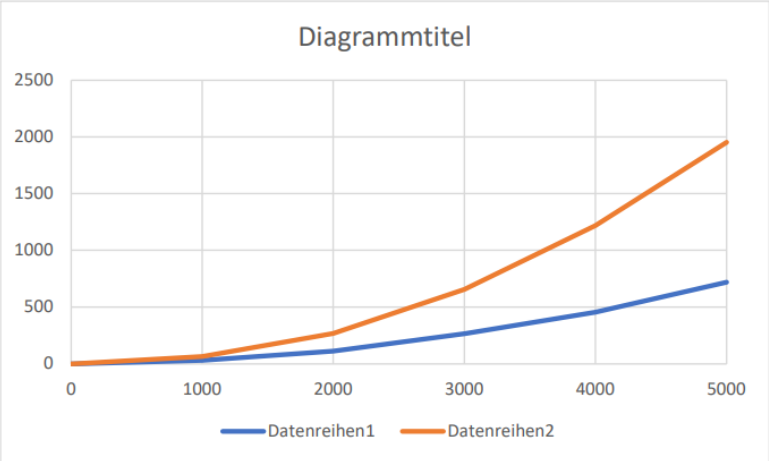
## Laufzeit und Wachstum von Sortieralgorithmen (KW 12)

**Minsort**

| Anzahl n | 0 | 5000 | 10000 | 15000 | 20000 | 25000 |
|---|---|---|---|---|---|---|
| Laufzeit ms | 0 | 30 | 111 | 264 | 454 | 719 |

**Bubblesort**

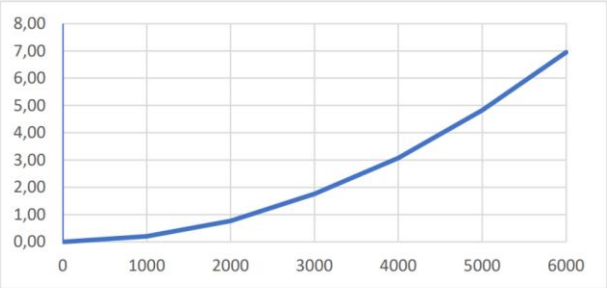
| Anzahl n | 0 | 5000 | 10000 | 15000 | 20000 | 25000 |
|---|---|---|---|---|---|---|
| Laufzeit ms | 0 | 63 | 266 | 656 | 1219 | 1954 |




In beiden Fällen handelt es sich um ein quadratisches Wachstum.
Begründung: Verdoppelt sich die Anzahl n der Zahlen, steigt die Rechenzeit um ca. den Faktor 4.
Bsp.: n= 10000 → 20000
Minsort: 111 ms → 454 ms; 111*4 ü. 454
Bubble: 266 ms → 1219 ms; 266*4 ü. 1219
Die Rechenzeit ist beim Bubblesort aber insgesamt schlechter, trotzdem die gleiche Klasse.

**Simple Sort**

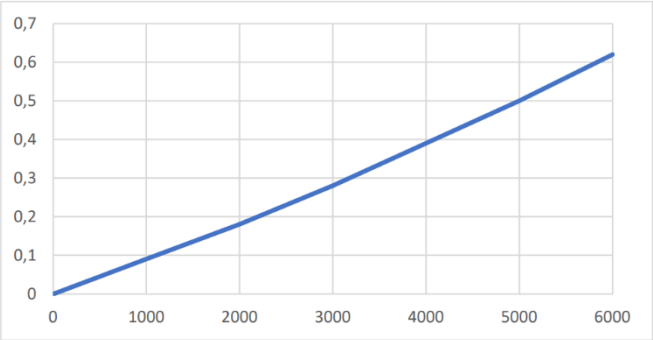
| Anzahl n | 0 | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 |
|---|---|---|---|---|---|---|---|
| Laufzeit ms | 0 | 0,203 | 0,768 | 1,761 | 3,073 | 4,82 | 6,957 |




Wachstum ist hier ebenfalls quadratisch.

**Counting-Sort**

| Anzahl n | 0 | 1000 | 2000 | 3000 | 4000 | 5000 | 6000 |
|---|---|---|---|---|---|---|---|
| Laufzeit ms | 0 | 0,09 | 0,18 | 0,28 | 0,39 | 0,5 | 0,62 |




Das Wachstum ist hier linear. Dies findet man sehr selten. Die wenigsten Sortieralgorithmen zeigen im durchschnittlichen Fall lineares Wachstum.

# O-Nation und Komplexität von Sortieralgorithmen (KW 13)

## Die O-Notation

Die **Laufzeit** eines Sortieralgorithmus meint die Art des Wachstums der

***Zuordnungsfunktion*: *Zeit --> Feldgröße***

In der Informatik interessiert dabei nicht die exakte Funktionsvorschrift, sondern nur die "Größenordnung" in Form der höchsten Potenz.
Die Notation hierfür nennt **sich O-Notation** (oder: Landau-Notation). Bsp.:

| **Funktion** | **Wachstum** | **O-Notation** |
|---|---|---|
| $5 \cdot n^2 + 4 \cdot n$ | quadratisch | O ($n^2$) |
| $\frac{4}{5}n + \frac{1}{2}$ | linear | O (n) |
| $15 \cdot 8^n + 2$ | exponentiell | O ($8^n$) |

Das Laufzeitverhalten von Algorithmen erschließt sich über die Anzahl der **Kernoperationen**.
Bei Sortieralgorithmen ist das z.B. die **Anzahl der Vergleiche oder Verschiebungen**.

## Übersicht zur Komplexität

Die Symbolik **O($n^2$)** bedeutet: Die Laufzeit ist ***höchsten*** asymptotisch proportional zu ***$n^2$***. Zur Angabe der Klasse wird immer der durchschnittliche Fall betrachtet, da ja Best-Case und Worst-Case nur extreme Grenzfälle darstellen.

| **Sortierverfahren** | **Best-Case** | **Average-Case** | **Worst-Case** | **Stabil** | **Zusätzlicher Speicherbedarf** |
|---|---|---|---|---|---|
| Selectionsort | **O($n^2$)** | **O($n^2$)** | **O($n^2$)** | nein | – |
| Insertionsort | **O(n)** | **O($n^2$)** | **O($n^2$)** | ja | – |
| Bubblesort | **O(n)** | **O($n^2$)** | **O($n^2$)** | ja | – |
| Shakersort | **O(n)** | **O($n^2$)** | **O($n^2$)** | ja | – |
| Heapsort | **O(n*log(n))** | **O(n*log(n))** | **O(n*log(n))** | nein | – |
| Quicksort | **O(n*log(n))** | **O(n*log(n))** | **O($n^2$)** | nein | **O(n*log(n))**für den Stack |

**Gebräuchliche Größenordnung**

| Größen-ordnung der Laufzeit T(n) | Verhalten der Laufzeit T(n), falls sich n verdoppelt | Bezeichnung |
|---|---|---|
| O(1) | ändert sich nicht | konstant |
| $O(\log_2 n)$ | wächst um eine kleine Konstante | logarithmisch |
| $O(\sqrt{n})$ | wächst um Faktor $\sqrt{2}$ | |
| O(n) | verdoppelt sich | linear |
| $O(n \log_2 n)$ | wird etwas mehr als doppelt so groß | |
| $O(n^2)$ | wächst um Faktor 4 | quadratisch |
| $O(n^3)$ | wächst um Faktor 8 | kubisch |
| $O(n^k)$ | wächst um Faktor $2^k$ | polynomiell |
| $O(2^n)$ | wächst sehr rasch | exponentiell |

## Bestimmung der Komplexität bei Selection- und Bubblesort

### 1. Bsp.: Selection-Sort (Minimum-Suche)

```
procedure select();
 const n=20;
 type vector=array[ 1 .. n] of integer;
 var   i, j, minposition, hilf: integer;
       a :  vector;
 begin
 for j:=1 to n-1 do
   begin
      minposition:=j;
      for i:=j+1 to n do
          if a[i] < a[minposition] then begin minposition:=i; end;
 hilf:= a[j];
 a[j] := a[minposition];
 a[minposition] := hilf ;
 end;
end;
```

Um ein Array mit n Einträgen mittels SelectionSort zu sortieren, muss **(n-1)** Mal das Minimum bestimmt und ebenso oft getauscht werden. Bei der ersten Bestimmung des Minimums sind n-1 Vergleiche notwendig, bei der zweiten n-2 Vergleiche usw. Mit der Gauß'schen Summenformel erhält man die Anzahl der notwendigen Vergleiche:

$$(n-1)+(n-2)+\ldots+3+2+1=\frac{(n-1)\cdot n}{2}=\frac{n^2}{2}-\frac{n}{2}$$

Man beachte, dass die exakte Schrittzahl dadurch, dass das erste Element **(n-1)** und nicht **n** ist, nicht genau der Darstellung der Gaußformel $n+(n-1)+\ldots+2+1=\frac{n\cdot(n+1)}{2}$ entspricht.

Aber, da ist nur einer Größeneinschätzung geht, liegt der SelectionSort in der Komplexitätsklasse **O($n^2$)**.

Da zum Ermitteln des Minimums immer der komplette noch nicht sortierte Teil des Arrays durchlaufen werden muss, liegt SelectionSort nicht nur im **Worst-Case**, sondern sogar in **jedem Fall** in dieser Komplexitätsklasse.

### 2. Bsp.: Bubble-Sort

```
procedure bubble();
 const n=20;
 type vector=array[ 1 .. n] of integer;
 var   i, j, hilf: integer;
       a: vector;
 begin
 for j:=1 to n-1 do
    for i:=1 to n- do
       if a[i] > a[i+1] then
        begin
          hilf:= a[i];
          a[i] := a[i+1];
         a[i+1] := hilf;
       end;
 end;
```

Bubblesort hat die Laufzeit ***O($n^2$)*** für Listen der Länge *n*. Im Falle der umgekehrt sortierten Liste (n, n-1, ..., 2, 1) werden maximal $\frac{n\cdot(n-1)}{2}$ viele Vertauschungen (Gauß'sche Summenformel) ausgeführt. Das entspricht wieder der Komplexitätsklasse ***O($n^2$)***.

Nur im **Best-Case** kann er eine Laufzeit von **O(n)** erreichen. Das ist der Fall, wenn der Array bereits von Beginn an nach dem Sortierkriterium sortiert ist. Das setzt aber die Implementierung eines Zeigers voraus, der ein erfasst, ob eine Vertauschung notwendig ist. Dies ist in der oberen Prozedur aber nicht der Fall.

## Aufgaben zu 2.

- Geben Sie die Komplexitätsklasse von Simple-Sort und Counting-Sort aus der letzten Übung an.

  | | | |
  |---|---|---|
  | Simple-Sort: | quadratisches Wachstum | O($n^2$) |
  | Counting-Sort: | lineares Wachstum | O(n) |

- Recherchieren Sie im Internet nach Beispielen von Sortieralgorithmen mit logarithmischem und exponentiellem Wachstum.

  Sortieralgorithmen mit logarithmischem Wachstum – $O(\log_2 n)$
  - Heap-Sort
  - Quicksort

  Sortieralgorithmen mit exponentiellen Wachstum – $O(2^n)$
  - Gibt keinen (expo. Wachstum ist sehr schlecht, maximal heute $O(n^2)$)

- Recherchieren Sie im Internet nach Beispielen von weiteren Algorithmen mit der Komplexitätsklasse $O(\sqrt{n})$ (Wurzel n) und $O(a^n)$ (exponentiell).

  ➔ Komplexitätsklasse Wurzel $O(\sqrt{n})$

  → Bsp. Primzahltest

  Hier wird eine ganze Zahl ***n*** geprüft, ob es eine Primzahl ist. Dafür gibt es keinen mathematischen Algorithmus. Die Vorgehensweise gestaltet sich meist wie folgt:
  - Berechnung der Quadratwurzel ***q*** und runden des Ergebnisses auf die nächst kleinere ganze Zahl ***(Begründung?)***
  - Prüfen, ob alle Primzahlen von 2 bis ***q*** Teiler von zu n teilerfremd sind
  - Findet sich ein Teiler, ist n keine Primzahl
  - Es kommen keine Werte größer als $\sqrt{n}$ in Frage, daher ist die Laufzeit beim Testen kleiner als bei linearem Wachstum

  ➔ Exponentiell → Fibonacci-Folge

  → Bsp. 1: Das **Problem des Handlungsreisenden** (Rundreiseproblem, engl. *Traveling Salesman Problem*) ist ein kombinatorisches Optimierungsproblem. Die Aufgabe besteht darin, eine Reihenfolge für den Besuch mehrerer Orte so zu wählen, dass keine Station außer der ersten mehr als einmal besucht wird, die gesamte Reisestrecke des Handlungsreisenden möglichst kurz und die erste Station gleich der letzten Station ist.

  → Bsp. 2: Die Ermittlung der **Fibonacci-Folge** bis zur Zahl n. Benannt ist die Folge nach Leonardo Fibonacci, der damit im Jahr 1202 das Wachstum einer Kaninchenpopulation beschrieb:

  *Ein Mann hält ein Kaninchenpaar an einem Ort, der gänzlich von einer Mauer umgeben ist. Wir wollen nun wissen, wie viele Paare von ihnen in einem Jahr gezüchtet werden können, wenn die Natur es so eingerichtet hat, dass diese Kaninchen jeden Monat ein weiteres Paar zur Welt bringen und damit im zweiten Monat nach ihrer Geburt beginnen.*

## 4. Der Sortieralgorithmus Insertion-Sort

**Insertion-Sort**

- Sortieren durch Einfügen

```
procedure insert();
  var
    i, j : integer;
    zeiger: integer;
 begin
    for i:=2 to n do
              begin
                zeiger:=a[i];
                j:=i;
                while (j>1) and (a[j-1]>zeiger) do begin a[j]:=a[j-1];dec(j);
             end;
       a[j]:=zeiger;
end;
```

**Das Verfahren**

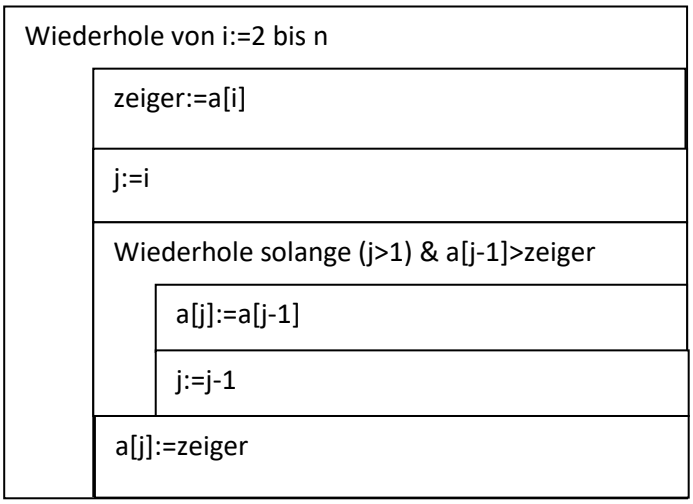
| | | a | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 33 | 27 | 39 | 19 | 30 | 22 | 21 | *unsortierte Zahlenfolge Zeiger eingekreist* |
| *i :=2* | j:=2 | | 33 | 33 | 39 | 19 | 30 | 22 | 21 | j>1 & a[1]>zeiger → a[2]:=a[1] |
| *i :=2* | j:=1 | | 27 | 33 | 39 | 19 | 30 | 22 | 21 | j=1→ Abbruch; a[1]:=zeiger |
| *i :=3* | j:=3 | | 27 | 33 | 39 | 19 | 30 | 22 | 21 | a[2]<zeiger; → Abbruch ;a[3]:=zeiger |
| *i :=4* | j:=4 | | 27 | 33 | 39 | 39 | 30 | 22 | 21 | j>1 & a[3]>zeiger → a[4]:=a[3] |
| *i :=4* | j:=3 | | 27 | 33 | 33 | 39 | 30 | 22 | 21 | j>1 & a[2]>zeiger → a[3]:=a[2] |
| *i :=4* | j:=2 | | 27 | 27 | 33 | 39 | 30 | 22 | 21 | j>1 & a[1]>zeiger → a[2]:=a[1] |
| *i :=4* | j:=1 | | 19 | 27 | 33 | 39 | 30 | 22 | 21 | j=1 → Abbruch; a[1]:=zeiger |
| *i :=5* | j:=5 | | 19 | 27 | 33 | 39 | 39 | 22 | 21 | j>1 & a[4]>zeiger → a[5]:=a[4] |
| *i :=5* | j:=4 | | 19 | 27 | 33 | 33 | 39 | 22 | 21 | j>1 & a[3]>zeiger → a[4]:=a[3] |
| *i :=5* | j:=3 | | 19 | 27 | 30 | 33 | 39 | 22 | 21 | j>1, aber a[2]<zeiger; Abbr.; a[3]:=zeiger |
| *i :=6* | j:=6 | | 19 | 27 | 30 | 33 | 39 | 39 | 21 | j>1 & a[5]>zeiger → a[6]:=a[5] |
| *i :=6* | j:=5 | | 19 | 27 | 30 | 33 | 33 | 39 | 21 | j>1 & a[4]>zeiger → a[5]:=a[4] |
| *i :=6* | j:=4 | | 19 | 27 | 30 | 30 | 33 | 39 | 21 | j>1 & a[3]>zeiger → a[4]:=a[3] |

## 1. Aufgabe

**Setzen Sie die Tabelle fort bis die Zahlen 21 und 22 an der richtigen Stelle stehen.**

| | | a | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | 19 | 27 | 30 | 33 | 39 | 22 | 21 | |
| *i :=6* | j:=4 | | 19 | 27 | 30 | 30 | 33 | 39 | 21 | j>1 & a[3]>zeiger → a[4]:=a[3] |
| *i :=6* | j:=3 | | 19 | 27 | 27 | 30 | 33 | 39 | 21 | j>1 & a[2]>zeiger → a[3]:=a[2] |
| *i :=6* | j:=2 | | 19 | 22 | 27 | 30 | 33 | 39 | 21 | j>1, aber a[2]<zeiger; Abbr.; a[2]:=zeiger |
| *i :=7* | j:=7 | | 19 | 22 | 27 | 30 | 33 | 39 | 39 | j>1 & a[6]>zeiger → a[7]:=a[6] |
| *i :=7* | j:=6 | | 19 | 22 | 27 | 30 | 33 | 33 | 39 | j>1 & a[5]>zeiger → a[6]:=a[5] |
| *i :=7* | j:=5 | | 19 | 22 | 27 | 30 | 30 | 33 | 39 | j>1 & a[4]>zeiger → a[5]:=a[4] |
| *i :=7* | j:=4 | | 19 | 22 | 27 | 27 | 30 | 33 | 39 | j>1 & a[3]>zeiger → a[4]:=a[3] |
| *i :=7* | j:=3 | | 19 | 22 | 22 | 27 | 30 | 33 | 39 | j>1 & a[2]>zeiger → a[3]:=a[2] |
| *i :=7* | j:=2 | | 19 | 21 | 22 | 27 | 30 | 33 | 39 | j>1, aber a[2]<zeiger; Abbr.; a[2]:=zeiger |

## 2. Vervollständigen Sie das Struktogramm zum Insertion-Sort.




## 3. Informieren Sie sich über die Komplexitätsklasse des Insertion-Sort

**Insertion-Sort gehört zur Komplexitätsklasse $O(n^2)$**

# 5. Der Sortieralgorithmus Quick-Sort

**Die Idee des Algorithmus „Quicksort“**

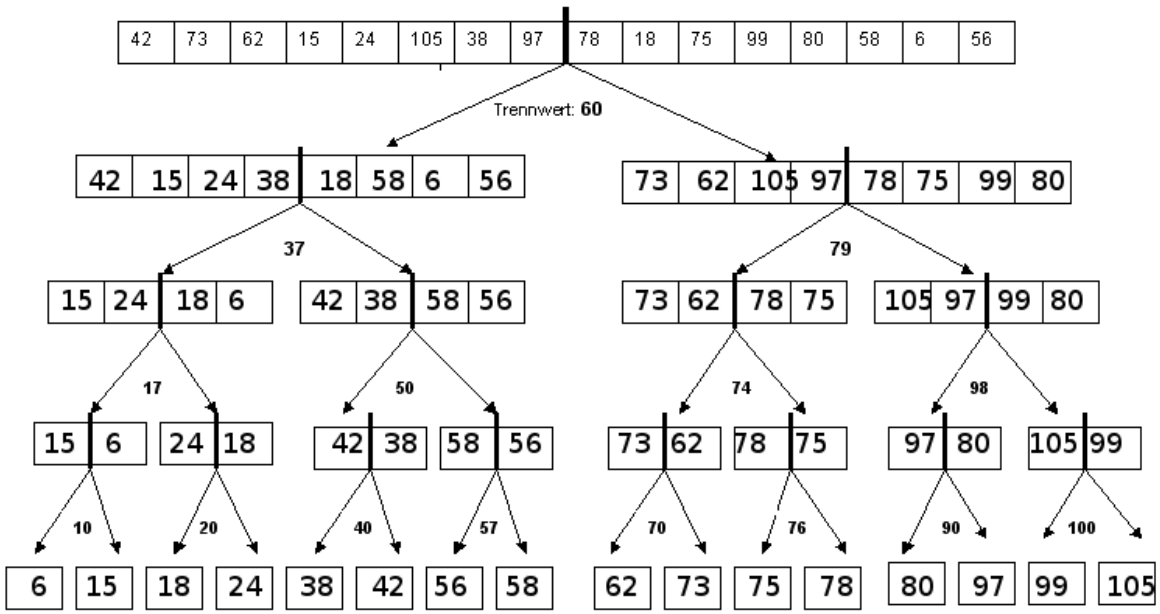
**Aufgabe**

Das Zahlenfeld 42 73 62 15 24 105 38 97 78 18 75 99 80 58 6 56 soll wie folgt sortiert werden: *Alle Zahlen, die kleiner oder gleich dem Trennwert 60 sind, landen in Feld **A1**, alle Zahlen die größer sind als der Trennwert 60 landen in Feld **A2**.*

| **A1** = {x \| x <= 60} | | **A2** = {x \| x > 60} |
|---|---|---|
| **42, 15, 24, 38, 18, 58, 6, 56** | **60** | **73, 62, 105, 97, 78, 75, 99, 80** |

**Aufgabe**

**Sortiere weiter!**

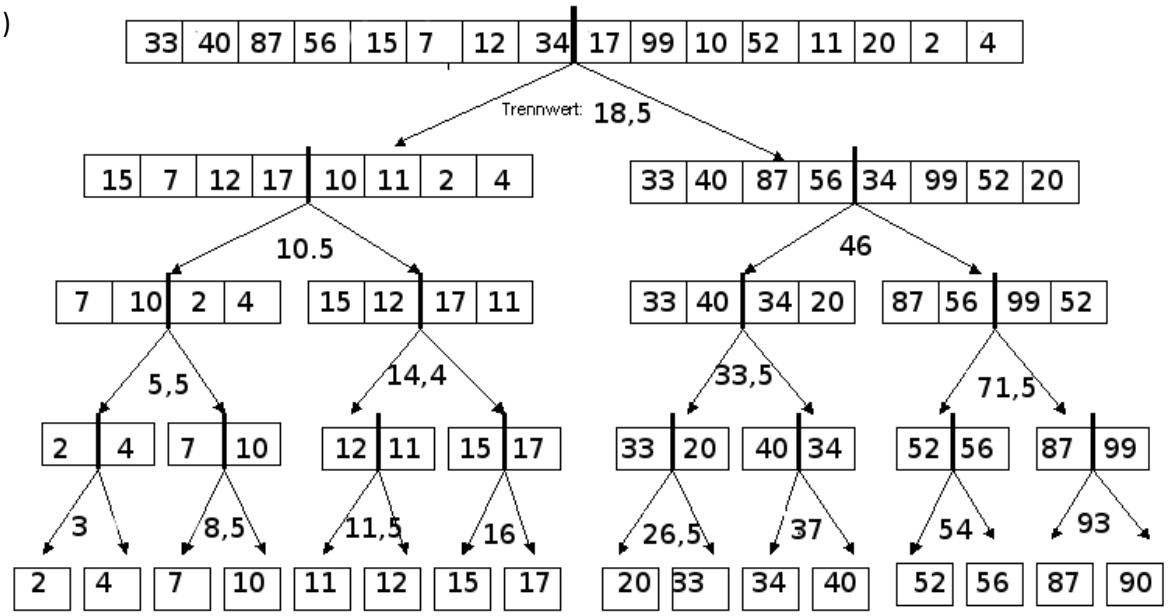


## 4. Sortieren Sie jetzt nachfolgende Zahlenlisten

Sortieren Sie die beiden Zahlenfelder nach „Quicksort“, wie es in der Idee vorgemacht wurde (Aufrufbaum). Einen geeigneten Trennwert müssen Sie jeweils selbst bestimmen.

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **a)** | 33 | 40 | 87 | 56 | 15 | 7 | 12 | 34 | 17 | 99 | 10 | 52 | 11 | 20 | 2 | 4 |
| **b)** | 48 | 43 | 7 | 54 | 40 | 23 | 56 | 1 | 91 | 39 | 87 | 5 | 86 | 31 | 99 | 39 |

a)




## 5. Geben Sie die Komplexitätsklasse des Quicksort an.

**Quick-Sort gehört zur Komplexitätsklasse O(n*log(n))**