Tutoraufgabe 1 (Sortieren):

a) Sortieren Sie das folgende Array durch Anwendung des Selectionsort-Algorithmus. Geben Sie dazu das Array nach jeder Swap-Operation an.

`[ 7,3,6,2,1 ]`

b) Sortieren Sie das folgende Array durch Anwendung des Insertionsort-Algorithmus. Geben Sie dazu das Array nach jeder Iteration der äußersten Schleife an.

`[ 7,2,3,6,1 ]`

c) Sortieren Sie das folgende Array durch Anwendung des Mergesort-Algorithmus. Geben Sie dazu das Eingabe-Array nach jeder Merge-Operation an.

`[ 4,7,2,9,1,8 ]`

Lösung:

a)

7 3 6 2 1

1 3 6 2 7

1 2 6 3 7

1 2 3 6 7

b)

7 2 3 6 1

2 7 3 6 1

2 3 7 6 1

2 3 6 7 1

1 2 3 6 7

c) Dieser Lösungsvorschlag enthält neben der ersten Zeile, die lediglich das Eingabe-Array zeigt, zusätzliche Zeilen (grau markiert), die zur Lösung der Aufgabe nicht nötig sind, sondern lediglich die Aufteilung des Arrays gemäß dem Mergesort-Algorithmus verdeutlichen sollen.

4 7 2 9 1 8

4 7 2 9 1 8

4 7 2 9 1 8

4 7 2 9 1 8

4 7 2 9 1 8

2 4 7 9 1 8

2 4 7 9 1 8

2 4 7 9 1 8

2 4 7 1 9 8

2 4 7 1 8 9

1 2 4 7 8 9

Aufgabe 2 (Sortieren): (3 + 3 + 8 = 14 Punkte)

a) Sortieren Sie das folgende Array durch Anwendung des Bubblesort-Algorithmus. Geben Sie dazu das Array nach jeder Swap-Operation an.

```
[ 6,2,3,7,1 ]
```

b) Sortieren Sie das folgende Array durch Anwendung des Quicksort-Algorithmus. Geben Sie dazu das Array nach jeder Partition-Operation an und markieren Sie das für die jeweilige Partitionierung genutzte Pivot-Element.

```
[ 4,7,2,8,9,6,1,3 ]
```

c) Sortieren Sie das folgende Array durch Anwendung des Heapsort-Algorithmus. Geben Sie dazu das Array nach jeder Swap-Operation an.

```
[ 4,7,2,3,9,6,1 ]
```

Lösung:

a)

6 2 3 7 1

2 6 3 7 1

2 3 6 7 1

2 3 6 1 7

2 3 1 6 7

2 1 3 6 7

1 2 3 6 7

b)

4 7 2 8 9 6 1 3

1 2 **3** 8 9 6 4 7

1 **2** 3 8 9 6 4 7

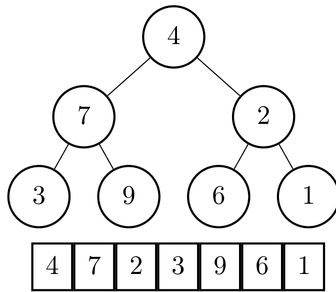
1 2 3 4 6 **7** 8 9

1 2 3 4 **6** 7 8 9

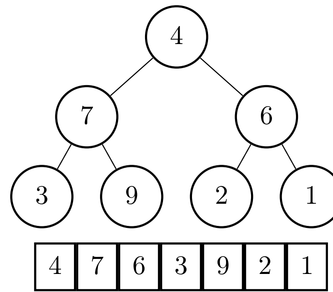
1 2 3 4 6 7 8 **9**

c) Der folgende Lösungsvorschlag enthält zusätzlich zu den Array-Zuständen auch ihre Interpretation als Heaps. Diese Angabe dient jedoch nur zur Erläuterung und war zur Lösung der Aufgabe nicht erforderlich.

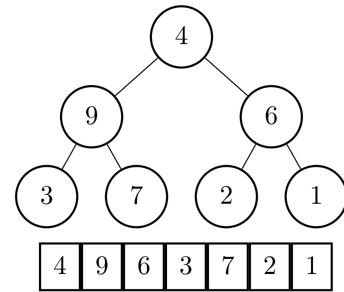
Schritt 0:



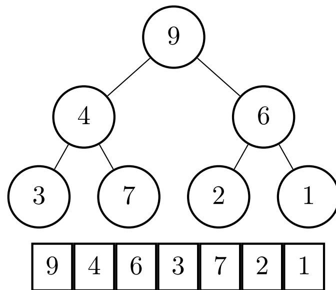
Schritt 1:



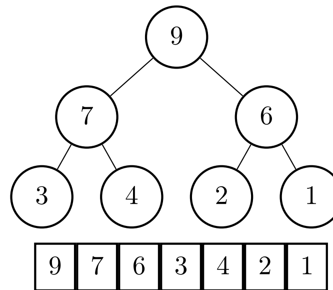
Schritt 2:



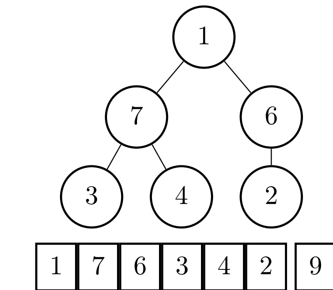
Schritt 3:



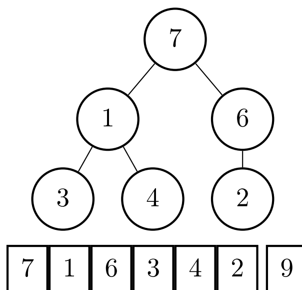
Schritt 4:



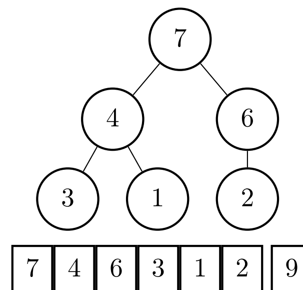
Schritt 5:



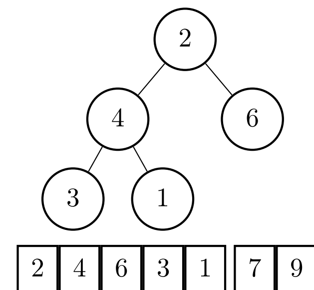
Schritt 6:



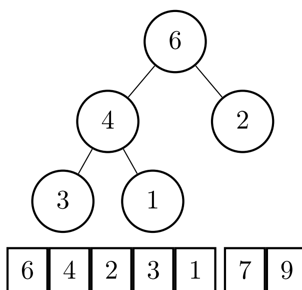
Schritt 7:



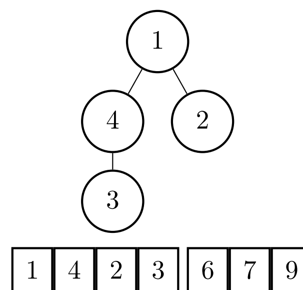
Schritt 8:



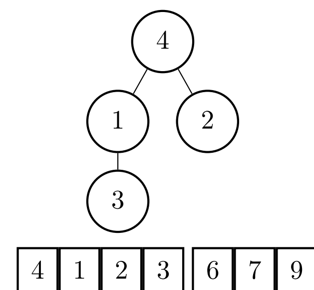
Schritt 9:



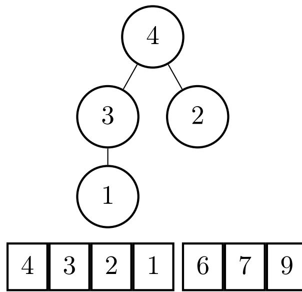
Schritt 10:



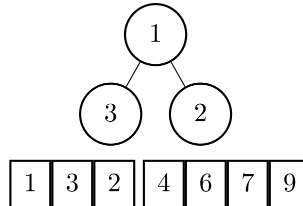
Schritt 11:



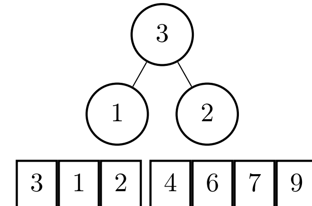
Schritt 12:



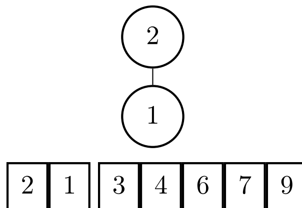
Schritt 13:



Schritt 14:



Schritt 15:



Schritt 16:

Tutoraufgabe 3 (Best- und Worstcase):

a) Geben Sie für ein beliebiges, aber festes $n \in \mathbb{N}$ ein Array der Länge n an, sodass der Bubblesort-Algorithmus so *viele* Swap-Operationen wie möglich benötigt (Worst-Case) und zusätzlich ein Array der Länge n, sodass der Bubblesort-Algorithmus so *wenig* Swap-Operationen wie möglich benötigt (Best-Case). Begründen Sie Ihre Antwort kurz.

b) Geben Sie für ein beliebiges, aber festes $n \in \mathbb{N}$ ein Array der Länge n an, sodass der Quicksort-Algorithmus mit der Pivot-Strategie aus der Vorlesung (als Pivot-Element wird immer das letzte Element des betrachteten Array-Bereichs gewählt) so *viele* Partition-Operationen wie möglich benötigt (Worst-Case). Begründen Sie Ihre Antwort kurz.

c) Geben Sie für ein beliebiges, aber festes $n \in \mathbb{N}$ ein Array der Länge n an, sodass der Heapsort-Algorithmus so *wenig* Swap-Operationen wie möglich benötigt (Best-Case). Begründen Sie Ihre Antwort kurz.

Lösung:

a) Worst-Case: $[n, n-1, \ldots, 1]$
Bei umgekehrter Sortierung wird in jedem Durchlauf der inneren Schleife das erste Element bis zum Index `lengthOfE` $- 1$ durchgetauscht (also `lengthOfE` $- 1$ viele Swap-Operationen) und der Wert von `lengthOfE` um 1 reduziert. Da dies die maximale Anzahl an Swap-Operationen pro Iteration und die maximale Anzahl an Iterationen liefert, ist dies der Worst-Case.

Best-Case: $[1, 2, \ldots, n]$
Bei vorsortierten Arrays werden überhaupt keine Swap-Operationen durchgeführt. Damit ist das offensichtlich der Best-Case.

b) Worst-Case: $[1, 2, \ldots, n]$
Bei vorsortierten Arrays führt die Wahl des letzten Elements als Pivot-Element dazu, dass in jeder Partition-Operation nur ein Element (nämlich das Pivot-Element selbst) abgespalten wird. Da man umso mehr Partition-Operationen benötigt, je weniger Elemente bei einer solchen Operation abgespalten werden, aber mindestens ein Element auf jeden Fall abgespalten wird, führt dies zum Worst-Case.

c) Best-Case: $\underbrace{[1, 1, \ldots, 1]}_{n}$

Bei Arrays mit ausschließlich gleichen Elementen wird die Heap-Eigenschaft nie verletzt. Damit führt der Algorithmus lediglich die Swap-Operationen in der `for`-Schleife von `heapSort` aus, die nicht vermieden werden können. Dies führt also offensichtlich zum Best-Case.

Aufgabe 4 (Best- und Worstcase): $(1 + 4 + 2 + 2^* = 7 + 2^*$ Punkte)

a) Geben Sie für ein beliebiges, aber festes $n \in \mathbb{N}$ ein Array der Länge n an, sodass der Selectionsort-Algorithmus so *viele* Swap-Operationen wie möglich benötigt (Worst-Case) und zusätzlich ein Array der Länge n, sodass der Selectionsort-Algorithmus so *wenig* Swap-Operationen wie möglich benötigt (Best-Case). Begründen Sie Ihre Antwort kurz.

b) Geben Sie für ein beliebiges, aber festes $n \in \mathbb{N}$ ein Array der Länge n an, sodass der Insertionsort-Algorithmus so *viele* Vergleiche wie möglich benötigt (Worst-Case) und zusätzlich ein Array der Länge n, sodass der Insertionsort-Algorithmus so *wenig* Vergleiche wie möglich benötigt (Best-Case). Begründen Sie Ihre Antwort kurz.

c) Geben Sie für ein beliebiges, aber festes $n \in \mathbb{N}$ ein Array der Länge n an, sodass der Mergesort-Algorithmus so *wenig* Vergleiche wie möglich benötigt (Best-Case). Begründen Sie Ihre Antwort kurz.

d)* Geben Sie für ein beliebiges, aber festes $n \in \mathbb{N}$ ein Array der Länge n an, sodass der Mergesort-Algorithmusso *viele* Vergleiche wie möglich benötigt (Worst-Case). Begründen Sie Ihre Antwort kurz.

Lösung:

a) Best- und Worst-Case: $[1, 2, \ldots, n]$
Für den in der Vorlesung vorgestellten Selectionsort-Algorithmus spielt der Inhalt des Eingabe-Arrays keine Rolle für die Anzahl an Swap-Operationen, da diese auf jeden Fall genau einmal in jeder Iteration der äußeren Schleife ausgeführt werden und die Anzahl der Iterationen nur von der Länge des Eingabe-Arrays abhängt.

b) Worst-Case: $[n, n-1, \ldots, 1]$
Bei umgekehrter Sortierung sind stets alle Elemente links vom aktuellen Element größer als dieses Element, wodurch das aktuelle Element ganz links eingefügt und die innere Schleife maximal oft durchlaufen wird. Da jede Iteration der inneren Schleife die Anzahl an Vergleichsoperationen erhöht und dies die einzige Möglichkeit ist, die Vergleichsanzahl zu erhöhen, führt dies offensichtlich zum Worst-Case.

Best-Case: $[1, 2, \ldots, n]$
Bei vorsortierten Arrays ist die Schleifenbedingung der inneren Schleife immer verletzt, sodass bei jeder Iteration der äußeren Schleife die minimale Anzahl von 3 Vergleichen durchgeführt wird. Da die Anzahl an Iterationen für die äußere Schleife unabhängig vom Array-Inhalt ist, führt dies offensichtlich zum Best-Case.

c) Best-Case: $[1, 2, \ldots, n]$
Bei vorsortierten Arrays werden stets alle Element des linken Arraybereichs vor dem rechten einsortiert. Damit erreicht man nie den dritten Fall und den zweiten Fall nur so oft wie nötig, um den jeweils linken Arraybereich einzusortieren. Dies führt zum Best-Case.

d)* **Anmerkung:** Die Lösung dieser Aufgabe geht von einer Annahme aus, die leider nicht im Vorfeld angegeben wurde: Vergleiche innerhalb einer `if`-Bedingung werden strikt ausgewertet, d. h. bei Vergleichen, die über `&&` oder `||` Operatoren verknüpft sind, werden alle Teilbedingungen ausgewertet, auch wenn das Gesamtergebnis für die Bedingung bereits nach Auswertung nur einer Teilbedingung feststehen würde. Ohne diese Annahme ist die Lösung für den Worst-Case wesentlich komplizierter, da man dann abwechselnd aus den beiden zu verschmelzenden Arraybereichen Elemente auswählen und das letzte Element

jeweils aus dem linken Teilarray stammen müsste. Die Abhängigkeit der Lösung von der Frage, ob Vergleiche strikt ausgewertet werden, war nicht beabsichigt und zukünftige Aufgaben werden so gestellt werden, dass ihre Lösungen unabhängig von dieser Frage sind.

Worst-Case: $[n, n-1, \ldots, 1]$
Die Anzahl an Vergleichen hängt davon ab, wie oft innerhalb der Merge-Operation der zweite oder dritte Fall erreicht wird. Bei umgekehrter Sortierung werden stets alle Elemente des rechten Arraybereichs vor dem linken einsortiert, wodurch man maximal oft den dritten Fall erreicht. Danach wird jedes Mal der zweite Fall erreicht, was auch zu einer maximalen Anzahl solcher Vergleiche und damit zum Worst-Case führt.

Tutoraufgabe 5 (Iteratives Sortieren):

Implementieren Sie für den Sortieralgorithmus `void mergeSort(int[] E, int left, int right)` eine iterative Variante `void mergeSortIter(int[] E, int left, int right)` mit einer Zeitkomplexität in $\mathcal{O}(n \log n)$, wobei Sie für Schreibzugriffe nur die Operation `merge(int[] E, int left, int mid, int right)` verwenden dürfen.
Nutzen Sie die Datenstruktur `Stack` mit den Operationen `void push(int v)`, `int pop()` und `bool isEmpty()`, um Rekursionen aufzulösen.
(`void push(int v)` schiebt `v` auf den Stack, `int pop()` entfernt den zuletzt eingefügten Wert vom Stack und gibt ihn zurück, `bool isEmpty()` gibt zurück, ob der Stack leer ist.)

Lösung:

Eine relativ unelegante Variante nutzt einen Stack, um die Rekursion aufzulösen. Dabei werden pro rekursivem Aufruf drei Elemente auf den Stack gepusht: Die zwei Grenzen des aktuellen Arrays, sowie eine Boolesche Variable (als `int` codiert), welche den Zustand des Algorithmus angibt (Splitting-Phase oder Merging-Phase).

```
void mergesort(int[] array, int left, int right)
{
  int[] stack;
  int mid = right/2;
  stack.push(left);
  stack.push(right);
  stack.push(1);
  stack.push(left);
  stack.push(mid);
  stack.push(0);
  stack.push(mid+1);
  stack.push(right);
  stack.push(0);

  while(!stack.isEmpty())
  {
    int sorted = stack.pop();
    int right = stack.pop();
    int left = stack.pop();
    if(sorted == 1)
    {
      // merge
      m = left + (right - left)/2;
      merge(array, left, m, right);
    }
    else
    {
```

```
        if (right - left > 0)
        {
          #split
          m = left + (right-left)/2;
          stack.push(left);
          stack.push(right);
          stack.push(1);
          stack.push(left);
          stack.push(m);
          stack.push(0);
          stack.push(m+1);
          stack.push(right);
          stack.push(0);
        }
      }
    }
}
```

Eine elegante iterative Lösung überspringt das rekursive Aufteilen des Arrays in gleiche Hälften und wendet `merge` bottom-up als wäre die Eingabelänge ein Potenz von 2 (d.h., die Längen der Teilsequenzen sind Potenzen von 2).
Ein Teilarray mit nur einem Element (Länge $k = 2^0 = 1$) ist automatisch sortiert. Beginnend am Anfang bei Position `left` wird zuerst `merge` auf Paaren von Teilsequenzen der Länge $k = 2^0 = 1$ angewandt, dann auf Teilsequenzen der Länge $k = 2^1 = 2$ usw., bis k die Länge der Eingabesequenz erreicht.

```
void mergeSortIter(int[] E, int left, int right)
{
  for (int k=1; k<=right-left; k = 2*k)}
    for (int i=left; i+k<=right; i=i+2*k){
      int left = i;
      int mid = i+k-1;
      int right = min(right, i+2*k-1);
      merge(E, left, mid, right);
    }
  }
}
```

Aufgabe 6 (Iteratives Sortieren): (6 Punkte)

Implementieren Sie für den Sortieralgorithmus `void quickSort(int[] E, int left, int right)` eine iterative Variante `void quickSortIter(int[] E, int left, int right)` mit einer Zeitkomplexität in $\mathcal{O}(n^2)$, wobei Sie für Schreibzugriffe nur die Operation `swap(int[] E, int i, int j)` (definiert als `int tmp = E[i]; E[i] = E[j]; E[j] = tmp;`) verwenden dürfen.
Nutzen Sie die Datenstruktur `Stack` mit den Operationen `void push(int v)`, `int pop()` und `bool isEmpty()`, um Rekursionen aufzulösen. (`void push(int v)` schiebt `v` auf den Stack, `int pop()` entfernt den zuletzt eingefügten Wert vom Stack und gibt ihn zurück, `bool isEmpty()` gibt zurück, ob der Stack leer ist.)

Lösung:

```
int[] quickSortIter(int[] E, int left, int right)
{
  Stack s = new Stack(2*(right-left));
  stack.push(left);
  stack.push(right);
```

```
    while(!s.isEmpty())
    {
      int right = stack.pop();
      int left = stack.pop();
      if (left < right)
      {
        int i = left;
        int j = right;
        int ppos = right;
        int pivot = E[ppos];
        while(true)
        {
          // advance until unsorted element found
          while(E[i] < pivot && i < right)
            i++;

          // reduce until unsorted element found
          while(E[j] >= pivot && j > left)
            j--;

          if(i >= j)
            break;

          swap(E, i, j);
        }
        swap(E, i, ppos);

        // recursive calls
        stack.push(left);
        stack.push(i-1);
        stack.push(i+1);
        stack.push(right);
      }
    }
  }
```

Bonusaufgabe 7 (Algorithmenanalyse*): (2*+4*=6* Punkte)

Gegeben sei folgender Algorithmus:

```
1  int foo(int[] E, int k) {
2      int left = 0;
3      List<int> m; // generiere eine leere Liste
4      while (left < E.length()) { // fuer jede Teilsequenz der Laenge 5
5          int right = left + 5;
6          if (right > E.length()) right = E.length();
7          int[] tmp = copyOfRange(E, left, right);  // kopiere E[left..right-1] nach tmp
8          sort(tmp); // sortiere tmp mit Insertionsort
9          if (E.length() <= 5) return tmp[k-1];
10         // nehme das mittlere Element aus tmp
11         // roundDown rundet zur naechsten ganzen Zahl ab
12         // add fuegt m das gegebene Element hinten an
13         m.add(tmp[roundDown(tmp.length()/2)]);
14         left = right;
15     }
16     // toArray erstellt zu einer Liste ein Array mit der gleichen Sequenz
17     p = foo(toArray(m), roundDown(m.length()/2));
```

```
18      List<int> E1, E2;
19      for (int i=0; i<E.length(); i++) {
20          if (i < p) E1.add(E[i]);
21          else if (i > p) E2.add(E[i]);
22      }
23      if (E1.length() + 1 == k) return p;
24      else if (E1.length() >= k) return foo(toArray(E1), k);
25      else return foo(toArray(E2), (k - E1.length() - 1));
26  }
```

a) Beschreiben Sie den Rückgabewert von `foo(E, k)`, falls `E` ein nicht leeres Array von paarweise verschiedenen ganzen Zahlen ist und `k` eine positive (> 0) ganze Zahl kleiner oder gleich der Länge von `E` ist. Begründen Sie Ihre Antwort!

b) Geben Sie die Rekursionsgleichungen $T(n,\ k)$ für die Worst-Case Laufzeit von `foo(E, k)` für den Fall an, dass $n =$ `E.length()` und $k =$ `k`. Dabei sind die elementaren Operationen `+`, `-`, `/`, `roundDown(..)`, `add(..)`, `length(..)` $\in \mathcal{O}(1)$. Außerdem ist `toArray(E)` $\in \mathcal{O}($`E.length()`$)$, `copyOfRange(E, b, r)` $\in \mathcal{O}(r - b)$ und `sort(tmp)` $\in \mathcal{O}(($`tmp.length()`$)^2)$. Beachten Sie, dass konstante Aufwände, die nicht in Verbindung mit einem rekursiven Aufruf stehen, durch eine Konstante c zusammengefasst werden können.

Lösung:

a) Der Algorithmus ist auch bekannt unter dem Namen *Median der Mediane* und findet das `k`-größte Element in `E`. Der Algorithmus sucht zunächst den Median von Teilsequenzen der Länge 5 und dann den Median p der gefundenen Mediane mittels rekursiven Aufrufs. Danach ist garantiert, dass jeweils die Hälfte der Elemente der Hälfte aller Teilsequenzen kleiner als p sind. Die Anzahl dieser Element lässt sich dann wie folgt bestimmen:

$$\frac{\frac{n}{5}}{2} \cdot 3 \;=\; \frac{3}{10} \cdot n$$

Im schlimmsten Fall befindet sich das gesuchte k-größte Element in den anderen $\frac{7}{10} \cdot n$ Elementen auf die der Algorithmus dann rekursiv aufgerufen wird.

b) Der Parameter k hat keinen Einfluss auf $T(n,\ k)$.

$$\begin{aligned}
T(n,\ \cdot) \quad = \quad & c_1 && |2-3: \text{ Initialisierung} \\
+ \quad & c_2 \cdot \tfrac{n}{5} && |4-15: \text{ Schleife} \\
+ \quad & c_3 + T(\tfrac{n}{5},\ \cdot) && |17: \text{ Rekursiver Aufruf auf den Medianen} \\
& && \quad \text{der Teilsequenzen in E mit Größe 5} \\
+ \quad & c_4 \cdot n && |18-22: \text{ Aufteilung der Elemente in E} \\
+ \quad & c_5 + T(\tfrac{7}{10}n,\ \cdot) && |23-25: \text{ Rekursiver Aufruf auf den Elementen} \\
& && \quad \text{in E die kleiner/größer als der Median sind} \\
= \quad & c_6 + c_7 \cdot n + T(\tfrac{n}{5},\ \cdot) + T(\tfrac{7}{10}n,\ \cdot)
\end{aligned}$$