## 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 |

## 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.

## 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);
      }
    }
  }
```