```java
try {
    File file = new File("figuren.txt");
    Scanner reader = new Scanner(file);

    while (reader.hasNextLine()) {
        String[] data = reader.nextLine().split(" ");
        switch (data[0]) {
            case "figuren.Rechteck" -> figuren.add(new Rechteck(Integer.parseInt(data[1]), Integer.parseInt(data[2]),
                    Integer.parseInt(data[3]), Integer.parseInt(data[4])));
            case "figuren.Kreis" -> figuren.add(new Kreis(Integer.parseInt(data[1]), Integer.parseInt(data[2]),
                    Integer.parseInt(data[3])));
            case "figuren.Linie" -> figuren.add(new Linie(Integer.parseInt(data[1]), Integer.parseInt(data[2]),
                    Integer.parseInt(data[3]), Integer.parseInt(data[4])));
        }
    }
    reader.close();
} catch (Exception ignored) { }
```

```java
try {
    FileWriter writer = new FileWriter("figuren.txt");

    StringBuilder sb = new StringBuilder();
    for (Figur f : figuren) {
        if (f instanceof Rechteck) {
            sb.append("figuren.Rechteck ").append(f.toString());
        } else if (f instanceof Kreis) {
            sb.append("figuren.Kreis ").append(f.toString());
        } else if (f instanceof Linie) {
            sb.append("figuren.Linie ").append(f.toString());
        }
        sb.append("\n");
    }
    writer.write(sb.toString());
    writer.close();
} catch (Exception ignored) { }
```

```java
for (int i = 1; i < array.length; i++) {
    if (array[(i - 1) / 2] < array[i])
        return false;
}
return true;
```

Für einen Knoten i gelten

1. PARENT(i) → return ⌊i/2⌋
2. LEFT(i) → return 2i
3. RIGHT(i) → return 2i + 1

```java
private void heapifyUp() {
    int index = size - 1;
    while (hasParent(index) && parent(index) < heap[index]) {
        swap(getParentIndex(index), index);
        index = getParentIndex(index);
    }
}
```

```java
public static boolean checkMinHeap(int[] A, int i)
{
    // if `i` is a leaf node, return true as every leaf node is a heap
    if (2*i + 2 > A.length) {
        return true;
    }

    // if `i` is an internal node

    // recursively check if the left child is a heap
    boolean left = (A[i] <= A[2*i + 1]) && checkMinHeap(A, 2*i + 1);

    // recursively check if the right child is a heap (to avoid the array index out
    // of bounds, first check if the right child exists or not)
    boolean right = (2*i + 2 == A.length) ||
            (A[i] <= A[2*i + 2] && checkMinHeap(A, 2*i + 2));

    // return true if both left and right child are heaps
    return left && right;
}
```

1. jede lebendige Zelle, die weniger als zwei lebendige Nachbarn hat, stirbt an Einsamkeit
2. jede lebendige Zelle mit mehr als drei lebendigen Nachbarn stirbt an Überbevölkerung
3. jede lebendige Zelle mit zwei oder drei Nachbarn fühlt sich wohl und lebt weiter
4. jede tote Zelle mit genau drei lebendigen Nachbarn wird wieder zum Leben erweckt

GameOfLife

```
machWas(int[] meinArray) {
    for (int i = 0; i < 5; i++) {
        meinArray[i] = 0;
    }
}
```

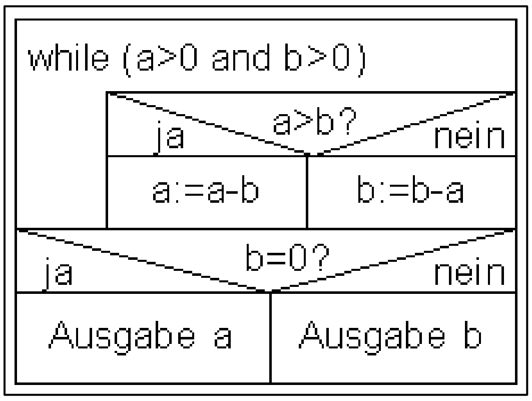
O(1)

```
machWas(int[] meinArray) {
    for (int i = 0; i < meinArray.length; i++) {
        for (int j = 0; j < mainArray.length; j++) {
            meinArray[i] = 0;
        }
    }
}
```

O(n^2)

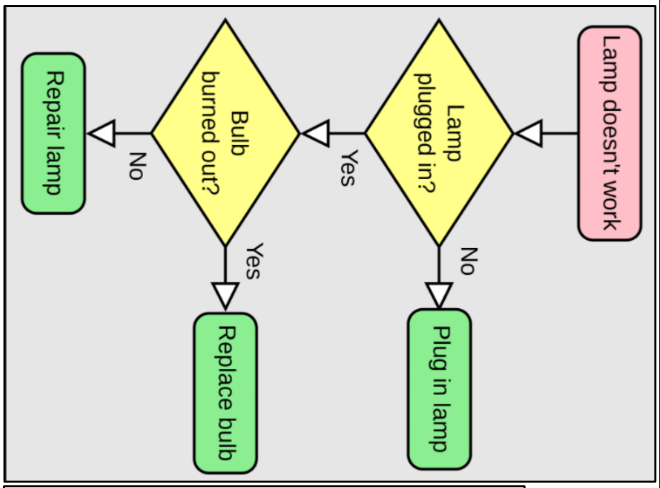
```
machWas(int[] meinArray) {
    for (int i = 0; i < meinArray.length; i++) {
        meinArray[i] = 0;
    }
}
```

O(n)



```
public static int[] sort(int[] arr) {
    boolean isSorted = false;
    while (!isSorted) {
        isSorted = true;
        for (int i = 0; i < arr.length - 1; i++) {
            if (arr[i] > arr[i + 1]) {
                int t = arr[i];
                arr[i] = arr[i + 1];
                arr[i + 1] = t;
                isSorted = false;
            }
        }
    }
    return arr;
}
```

BubbleSort



```
start = System.currentTimeMillis();
```

```
public static int[] sort(int[] arr) {
    int l = arr.length;
    for (int i = 1; i < l; i++) {
        int k = arr[i];
        for (int j = i - 1; j >= 0; j--) {
            if (arr[j] > k) {
                arr[j + 1] = arr[j];
                arr[j] = k;
            }
        }
    }
    return arr;
}
```

InsertionSort

```
void sort(int arr[])
{
    int n = arr.length;

    // One by one move boundary of unsorted subarray
    for (int i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        int min_idx = i;
        for (int j = i+1; j < n; j++)
            if (arr[j] < arr[min_idx])
                min_idx = j;

        // Swap the found minimum element with the first
        // element
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
```

SelectionSort