
Algorithmen und Datenstrukturen

Codemonkeys Lösungen von Fabian Damken



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Inhaltsverzeichnis

1	Lizenz	5
2	Einführung	6
3	Lösungen	7
3.1	Array List	8
3.1.1	Contains	8
3.1.2	Insert at Position	8
3.1.3	Remove	8
3.2	Priority Queue	8
3.2.1	Peek	8
3.2.2	Push	9
3.2.3	Pop	9
3.3	Array	9
3.3.1	Array is Sorted	9
3.3.2	Binary Search Iterative	10
3.3.3	Binary Search Recursive	10
3.3.4	Insert	10
3.3.5	Insert at Index	10
3.3.6	Quick-Sort Recursive	11
3.3.7	Remove	11
3.3.8	Search second largest Element	11
3.3.9	Sort $O(n^2)$ Iterative	12
3.3.10	Duplicate every second Element	12
3.3.11	Linear Search	12
3.3.12	Merge	13
3.3.13	Rotate Pairs	13
3.3.14	Rotate successive Triples	13
3.3.15	Rotate Tripels	14
3.3.16	Selection-Sort Iterative	14
3.3.17	Ehift elements left	14
3.3.18	Shift elements right	15
3.4	Baum	15
3.4.1	Binary Search Tree: Add	15
3.4.2	Binary Search Tree: Remove	15
3.4.3	Binary Search Tree: Traverse	15
3.5	Graph	16
3.5.1	AStern: Complete	16
3.5.2	AStern: Break Condition/Variant	16
3.5.3	AStern: Functionality	16
3.5.4	Bellman-Ford: Complete	16
3.5.5	Bellman-Ford: Break Condition/Variant	16
3.5.6	Bellman-Ford: Functionality	16
3.5.7	Dijkstra: Complete	16
3.5.8	Dijkstra: Break Condition/Variant	16

3.5.9	Dijkstra: Functionality	16
3.5.10	Dijkstra: Invariant	17
3.5.11	Floyd-Warshall: Complete	17
3.5.12	Floyd-Warshall: Break Condition/Variant	17
3.5.13	Floyd-Warshall: Functionality	17
3.5.14	Add Edge	17
3.5.15	Add Node	18
3.5.16	Add Subgraph	18
3.5.17	Count Edges	18
3.5.18	Count Nodes	18
3.5.19	Find connected Subgraphs	19
3.5.20	Find Node	19
3.5.21	Remove Edge	19
3.5.22	Remove Node	19
3.5.23	Kruskal: Complete	20
3.5.24	Kruskal: Union Find	20
3.5.25	Kruskal: Break Condition/Variant	20
3.5.26	Kruskal: Functionality	20
3.5.27	Kruskal: Invariant	20
3.5.28	Prim: Complete	20
3.5.29	Prim: Break Condition/Variant	20
3.5.30	Prim: Functionality	20
3.5.31	Prim: Invariante	21
3.6	Iterativ	21
3.6.1	Palindrome Check	21
3.6.2	Fibonacci	21
3.6.3	Multiple Palindrome Check	21
3.7	MIPS	22
3.7.1	Sort Array	22
3.7.2	Euklidischer Algorithmus	22
3.7.3	Pascalsches Dreieck	22
3.8	Rekursiv	22
3.8.1	Approximate Square Root	22
3.8.2	Fibonacci	22
3.9	Single Linked List	23
3.9.1	Insert First (single)	23
3.9.2	Insert Last (single)	23
3.9.3	Clone Elements	23
3.9.4	Clone List	24
3.9.5	Duplicate every second Element	24
3.9.6	Get at Index	24
3.9.7	Insert	25
3.9.8	Insert First	25
3.9.9	Insert Last	26
3.9.10	Insert (single)	26
3.9.11	Invert	27
3.9.12	Merge Linked Lists	27
3.9.13	Remove	28
3.9.14	Remove First	28
3.9.15	Remove Last	28

3.9.16	Remove duplicated Elements	29
3.10	String Operations	29
3.10.1	Prefix Check	29
3.10.2	Simple String Matcher	30
3.11	Lambda	30
3.11.1	Lambda Expressions/Strategy Pattern	30

Todo list

Array List: Insert at Position	8
Array: Binary Search Iterative	10
Array: Binary Search Recursive	10
Array: Quick-Sort Recursive	11
Baum: Binary Search Tree: Add	15
Baum: Binary Search Tree: Remove	15
Graph: AStern: Complete	16
Graph: AStern: Break Condition/Variant	16
Graph: AStern: Functionality	16
Graph: Bellman-Ford: Complete	16
Graph: Bellman-Ford: Break Condition/Variant	16
Graph: Bellman-Ford: Functionality	16
Graph: Dijkstra: Complete	16
Graph: Dijkstra: Break Condition/Variant	16
Graph: Dijkstra: Functionality	16
Graph: Dijkstra: Invariant	17
Graph: Floyd-Warshall: Complete	17
Graph: Floyd-Warshall: Break Condition/Variant	17
Graph: Floyd-Warshall: Functionality	17
Graph: Add Subgraph	18
Graph: Count Nodes	18
Graph: Find connected Subgraphs	19
Graph: Kruskal: Complete	20
Graph: Kuskal: Union Find	20
Graph: Kruskal: Break Condition/Variant	20
Graph: Kruskal: Functionality	20
Graph: Kuskal: Invariant	20
Graph: Prim: Complete	20
Graph: Prim: Break Condition/Variant	20
Graph: Prim: Functionality	20
Graph: Prim: Invariante	21
MIPS: Sort Array	22
MIPS: Euklidischer Algorithmus	22
MIPS: Pascalsches Dreieck	22
Single Linked List: Insert	25
Single Linked List: Remove	28

1 Lizenz

Bei Weitergabe dieses Dokumentes ist es obligatorisch, dass der Lizenzgeber in angemessener Form genannt wird. Außerdem muss kenntlich gemacht werden, ob das Dokument verändert wurde. Das Dokument darf nur unter den selben Bedingungen wie das Original weitergegeben werden.

Eine Nennung des Lizenzgebers wie in diesem Dokument gilt als angemessen.

Die Nutzung des Codes in diesem Dokument ist ohne weitere Bedingungen zulässig. Es ist sich an die Plagiarismusregelungen des Fachbereiches Informatik der TU Darmstadt zu halten. Diese sind unter <https://www.informatik.tu-darmstadt.de/de/studierende/studium/plagiarismus/> nachzulesen.

2 Einführung

Dieses Dokument enthält (selbsterstellte!) Lösungen für Codemonkeys.

3 Lösungen

Die folgenden Sektionen erhalten Lösungen für Codemonkeys,

Warnung: Bei Aufgaben mit mehreren Methoden muss jede Methode einzeln an den Server gesendet werden, da die Änderungen sonst nicht übernommen werden. Ob die Version auf dem Server aktuell ist, ist mit einem kleinen Kreuz (= nicht aktuell) oder einem kleinen Häkchen (= aktuell) im Panelkopf der Methode gekennzeichnet.

3.1 Array List

3.1.1 Contains

```
1 {
2     for (ArrayListElement<T> el = getFirst();
3         el != null; el = el.getNext()) {
4         for (Listobject<T> obj : el.getData()) {
5             if (obj != null && data.equals(obj.getData())) {
6                 return true;
7             }
8         }
9     }
10    return false;
11 }
```

3.1.2 Insert at Position

3.1.3 Remove

```
1 {
2     if (i < 0 || getFirst() == null) {
3         return false;
4     }
5
6     int index = 0;
7     for (ArrayListElement<T> el = getFirst();
8         el != null; el = el.getNext()) {
9         final Listobject<T>[] data = el.getData();
10        boolean found = false;
11        for (int j = 0; j < el.getN(); j++) {
12            if (index > i) {
13                data[j - 1] = data[j];
14            } else if (index == i) {
15                found = true;
16            }
17
18            index++;
19        }
20        if (found) {
21            el.decN();
22            return true;
23        }
24    }
25    return false;
26 }
```

Array
List:
In-
sert
at
Po-
si-
ti-
on

3.2 Priority Queue

3.2.1 Peek

```
1 {
2     return getHead() == null ? null : getHead().getKey();
3 }
```

3.2.2 Push

```
1 {
2     if (key == null) {
3         return false;
4     }
5
6     final MListElement<T> elem = new MListElement<T>(key);
7     final MListElement<T> head = getHead();
8     if (head == null) {
9         setHead(elem);
10
11        return true;
12    }
13    if (getComp().compare(key, head.getKey()) <= 0) {
14        elem.setNext(head);
15        setHead(elem);
16
17        return true;
18    }
19
20    for (MListElement<T> el = getHead(); el != null; el = el.getNext()) {
21        if (getComp().compare(key, el.getKey()) > 0
22            && (el.getNext() == null
23                || getComp().compare(key, el.getNext().getKey()) <= 0)) {
24            elem.setNext(el.getNext());
25            el.setNext(elem);
26
27            return true;
28        }
29    }
30    return false;
31 }
```

3.2.3 Pop

```
1 {
2     final MListElement<T> head = getHead();
3     if (head == null) {
4         return null;
5     }
6     setHead(head.getNext());
7     return head.getKey();
8 }
```

3.3 Array

3.3.1 Array is Sorted

```

1 {
2     if (a == null) {
3         return false;
4     }
5
6     for (int i = 0; i < a.length - 1; i++) {
7         Integer x = a[i];
8         Integer y = a[i + 1];
9         if (x == null || y == null || comp.compare(x, y) > 0) {
10            return false;
11        }
12    }
13    return true;
14 }

```

3.3.2 Binary Search Iterative

3.3.3 Binary Search Recursive

3.3.4 Insert

```

1 {
2     final Listobject<T>[] oldArray = getArray();
3     final Listobject<T>[] newArray = new Listobject[oldArray.length + 1];
4     boolean inserted = false;
5     for (int i = 0; i < newArray.length; i++) {
6         if (inserted) {
7             newArray[i] = oldArray[i - 1];
8         } else if (i == oldArray.length) {
9             newArray[i] = element;
10        } else {
11            final Listobject<T> el = oldArray[i];
12            if (el.compareTo(element) < 0) {
13                newArray[i] = el;
14            } else if (el.compareTo(element) > 0) {
15                newArray[i] = element;
16                inserted = true;
17            } else {
18                newArray[i] = element;
19                inserted = true;
20            }
21        }
22    }
23    setArray(newArray);
24    return newArray;
25 }

```

Array:
Bi-
na-
ry
Search
Ite-
ra-
ti-
ve

Array:
Bi-
na-
ry
Search
Re-
cur-
si-
ve

3.3.5 Insert at Index

```

1 {
2     final Listobject<T>[] oldArray = this.getArray();

```

```

3     final Listobject<T>[] newArray;
4     if (pos < oldArray.length && oldArray[pos] == null) {
5         newArray = new Listobject[oldArray.length];
6         for (int i = 0; i < oldArray.length; i++) {
7             newArray[i] = oldArray[i];
8         }
9         newArray[pos] = element;
10    } else {
11        newArray = new Listobject[pos < oldArray.length
12            ? oldArray.length + 1
13            : pos + 1];
14        for (int i = 0; i < newArray.length; i++) {
15            if (i < pos && i < oldArray.length) {
16                newArray[i] = oldArray[i];
17            } else if (i > pos && i <= oldArray.length) {
18                newArray[i] = oldArray[i - 1];
19            } else if (i == pos) {
20                newArray[i] = element;
21            }
22        }
23    }
24    setArray(newArray);
25    return newArray;
26 }

```

3.3.6 Quick-Sort Recursive

3.3.7 Remove

```

1 {
2     final Listobject<T>[] newArray = new Listobject[array.length - 1];
3     for (int i = 0; i < array.length; i++) {
4         if (i < index) {
5             newArray[i] = array[i];
6         } else if (i > index) {
7             newArray[i - 1] = array[i];
8         }
9     }
10    return newArray;
11 }

```

Array:
Quick-
Sort
Re-
cur-
si-
ve

3.3.8 Search second largest Element

```

1 {
2     for (int i = 0; i < getLength(); i++) {
3         if (getElem(i) == null) {
4             continue;
5         }
6
7         if (getLargest() == -1) {
8             setLargest(i);
9         } else if (getComp().compare(
10            getElem(i), getElem(getLargest())) >= 0) {
11            if (getComp().compare(getElem(i), getElem(getLargest())) != 0) {

```

```

12         setSecLargest(getLargest());
13     }
14     setLargest(i);
15 } else if (getSecLargest() == -1
16           || getComp().compare(getElem(i),
17                                getElem(getSecLargest())) >= 0) {
18     setSecLargest(i);
19 }
20 }
21 }

```

3.3.9 Sort $O(n^2)$ Iterative

```

1 {
2     for (int i = 0; i < inputdata.length; i++) {
3         for (int j = 1; j < inputdata.length - i; j++) {
4             if (comp.compare(inputdata[j - 1], inputdata[j]) > 0) {
5                 final Listobject<T> tmp = inputdata[j - 1];
6                 inputdata[j - 1] = inputdata[j];
7                 inputdata[j] = tmp;
8             }
9         }
10    }
11
12    return inputdata;
13 }

```

3.3.10 Duplicate every second Element

```

1 {
2     final Listobject<T>[] result =
3         new Listobject[array.length + array.length / 2];
4     for (int i = 0, j = 0; i < array.length; i++) {
5         result[j++] = array[i];
6         if (i % 2 != 0) {
7             result[j++] = array[i];
8         }
9     }
10    return result;
11 }

```

3.3.11 Linear Search

```

1 {
2     if (getElem() == null) {
3         return -1;
4     }
5
6     for (int i = 0; i < getArrayLength(); i++) {
7         if (getArrayElem(i) == null) {
8             continue;
9         }
10
11        if (getComp().compare(getElem(), getArrayElem(i)) == 0) {
12            return i;

```

```
13     }
14 }
15 return -1;
16 }
```

3.3.12 Merge

```
1 {
2     final Listobject<T>[] result =
3         new Listobject[left.length + right.length];
4     int i = 0;
5     int a = 0;
6     int b = 0;
7     for (; a < left.length && b < right.length; i++) {
8         final Listobject<T> aElem = left[a];
9         final Listobject<T> bElem = right[b];
10        if (aElem.compareTo(bElem) < 0) {
11            result[i] = aElem;
12            a++;
13        } else {
14            result[i] = bElem;
15            b++;
16        }
17    }
18    for (; a < left.length; i++, a++) {
19        result[i] = left[a];
20    }
21    for (; b < right.length; i++, b++) {
22        result[i] = right[b];
23    }
24    return result;
25 }
```

3.3.13 Rotate Pairs

```
1 {
2     if (list == null) {
3         throw new NullPointerException();
4     }
5
6     for (int i = 1; i < list.length; i += 2) {
7         final Listobject<T> tmp = list[i - 1];
8         list[i - 1] = list[i];
9         list[i] = tmp;
10    }
11
12    return list;
13 }
```

3.3.14 Rotate successive Triples

```
1 {
2     if (list == null) {
3         throw new IllegalArgumentException();
4     }
5 }
```

```

5
6     for (int i = 2; i < list.length; i += 3) {
7         final Listobject<T> a = list[i - 2];
8         final Listobject<T> b = list[i - 1];
9         final Listobject<T> c = list[i];
10
11         list[i - 2] = b;
12         list[i - 1] = c;
13         list[i] = a;
14     }
15     return list;
16 }

```

3.3.15 Rotate Tripels

```

1 {
2     if (a < 0 || b < 0 || c < 0) {
3         throw new IndexOutOfBoundsException();
4     }
5     if (a >= array.length || b >= array.length || c >= array.length) {
6         return false;
7     }
8
9     final Listobject<T> aElem = array[a];
10    final Listobject<T> bElem = array[b];
11    final Listobject<T> cElem = array[c];
12    array[a] = cElem;
13    array[b] = aElem;
14    array[c] = bElem;
15
16    return true;
17 }

```

3.3.16 Selection-Sort Iterative

```

1 {
2     for (int i = array.length - 1; i > 0; i--) {
3         int m = 0;
4         for (int j = 0; j < i; j++) {
5             if (array[m].compareTo(array[j]) < 0) {
6                 m = j;
7             }
8         }
9
10        if (array[m].compareTo(array[i]) > 0) {
11            Listobject<T> tmp = array[i];
12            array[i] = array[m];
13            array[m] = tmp;
14        }
15    }
16    return array;
17 }

```

3.3.17 Ehift elements left

```

1 {
2     if (list == null || list.length == 0) {
3         throw new IllegalArgumentException();
4     }
5
6     Listobject<T> first = list[0];
7     for (int i = 0; i < list.length - 1; i++) {
8         list[i] = list[i + 1];
9     }
10    list[list.length - 1] = first;
11
12    return list;
13 }

```

3.3.18 Shift elements right

```

1 {
2     if (list == null) {
3         return null;
4     }
5
6     Listobject<T>[] result = Listobject
7         .factoryMethodListobjectTArray(list.length);
8     result[0] = list[list.length - 1];
9     for (int i = 0; i < list.length - 1; i++) {
10        result[i + 1] = list[i];
11    }
12    return result;
13 }

```

3.4 Baum

3.4.1 Binary Search Tree: Add

3.4.2 Binary Search Tree: Remove

3.4.3 Binary Search Tree: Traverse

getElements

```

1 {
2     final ArrayList<N> data = new ArrayList<>();
3     if (getRoot() != null) {
4         getElementsRec(getRoot(), data);
5     }
6     return data;
7 }

```

getElementsRec

Baum:
Bi-
na-
ry
Search
Tree:
Add

Baum:
Bi-
na-
ry
Search
Tree:
Re-
mo-
ve


```

1  {
2      final Node<N, Integer> left = getLeft(node);
3      final Node<N, Integer> right = getRight(node);
4      if (left != null) {
5          getElementsRec(left, visited);
6      }
7      visited.add(node.getData());
8      if (right != null) {
9          getElementsRec(right, visited);
10     }
11 }

```

3.5 Graph

3.5.1 AStern: Complete

3.5.2 AStern: Break Condition/Variant

3.5.3 AStern: Functionality

3.5.4 Bellman-Ford: Complete

3.5.5 Bellman-Ford: Break Condition/Variant

3.5.6 Bellman-Ford: Functionality

3.5.7 Dijkstra: Complete

3.5.8 Dijkstra: Break Condition/Variant

3.5.9 Dijkstra: Functionality

Graph:
AS-
tern:
Com-
ple-
te

Graph:
AS-
tern:
Break
Con-
di-
tion/Va-
ri-
ant

Graph:
AS-
tern:
Func-
tio-
na-
li-
ty

Graph:
Bellman
Ford:
Com-
ple-
te

Graph:
Bellman
Ford:
Break
Con-
di-
tion/Va-
ri-
ant

3.5.10 Dijkstra: Invariant

3.5.11 Floyd-Warshall: Complete

3.5.12 Floyd-Warshall: Break Condition/Variant

3.5.13 Floyd-Warshall: Functionality

3.5.14 Add Edge

Methode 1

```
1 {
2     if (from == null || to == null || data == null) {
3         throw new IllegalArgumentException();
4     }
5     if (getFanOutMax() < from.getFanOut().size() + 1 || getFanInMax() <
6         to.getFanIn().size() + 1) {
7         throw new FanOverflowException("");
8     }
9     final Edge<N, E> edge = new Edge(from, to, data);
10    from.getFanOut().add(edge);
11    to.getFanIn().add(edge);
12    final ArrayList<Edge<N, E>> edges = getEdgeList();
13    edges.add(edge);
14    setEdgeList(edges);
15 }
```

Methode 2

```
1 {
2     if (from == null || to == null || data == null) {
3         throw new IllegalArgumentException();
4     }
5     if (getFanOutMax() <= from.getFanOut().size() + 1 || getFanInMax() <=
6         to.getFanIn().size() + 1) {
7         throw new FanOverflowException("");
8     }
9     final Edge<N, E> edge = new Edge(from, to, data);
10    from.getFanOut().add(edge);
11    to.getFanIn().add(edge);
12    final ArrayList<Edge<N, E>> edges = getEdgeList();
13    edges.add(edge);
14    setEdgeList(edges);
15 }
```

Graph:
Di-
jkstra:

In-
va-
ri-
ant

Graph:
Floyd-
Warsha
Com-
ple-
te

Graph:
Floyd-
Warsha
Break
Con-
di-
tion/Va-
ri-
ant

Graph:
Floyd-
Warsha
Func-
tio-
na-
li-
ty

3.5.15 Add Node

```
1 {
2     if (data == null) {
3         return null;
4     }
5
6     final Node<N, E> node = new Node(getIdGen(), data);
7     final ArrayList<Node<N, E>> nodes = getNodeList();
8     nodes.add(node);
9     setNodeList(nodes);
10    return node;
11 }
```

3.5.16 Add Subgraph

3.5.17 Count Edges

countEdgesRec

```
1 {
2     if (!nodeSet.add(node)) {
3         return;
4     }
5
6     for (final Edge<N, E> edge : node.getFanOut()) {
7         edgeSet.add(edge);
8
9         countEdgesRec(edge.getTargetNode(), edgeSet, nodeSet);
10    }
11    for (final Edge<N, E> edge : node.getFanIn()) {
12        edgeSet.add(edge);
13
14        countEdgesRec(edge.getTargetNode(), edgeSet, nodeSet);
15    }
16 }
```

countEdgesInConnectedGraph

```
1 {
2     if (node == null || !contains(node)) {
3         return -1;
4     }
5
6     final HashSet<Edge<N, E>> edges = new HashSet<>();
7     final HashSet<Node<N, E>> nodes = new HashSet<>();
8     countEdgesRec(node, edges, nodes);
9     return edges.size();
10 }
```

3.5.18 Count Nodes

Graph:
Add
Sub-
graph

Graph:
Count
No-
des

3.5.19 Find connected Subgraphs

3.5.20 Find Node

```
1 {
2   if (startNode == null || data == null || !contains(startNode)) {
3     return null;
4   }
5
6   final HashSet<Node<N, E>> processed = new HashSet<>();
7   final ArrayDeque<Node<N, E>> stack = new ArrayDeque<>();
8   stack.push(startNode);
9   while (!stack.isEmpty()) {
10    final Node<N, E> node = stack.pop();
11
12    if (processed.contains(node)) {
13      continue;
14    }
15
16    processed.add(node);
17    for (final Edge<N, E> edge : node.getFanOut()) {
18      stack.push(edge.getTargetNode());
19    }
20
21    if (objectEquals(node.getData(), data)) {
22      return node;
23    }
24  }
25  return null;
26 }
```

Graph:
Find
connec-
ted
Sub-
graphs

3.5.21 Remove Edge

```
1 {
2   if (edge == null || !getEdgeList().contains(edge)) {
3     return false;
4   }
5
6   final ArrayList<Edge<N, E>> edges = getEdgeList();
7   edges.remove(edge);
8   setEdgeList(edges);
9
10  edge.removeFromNodes();
11
12  return true;
13 }
```

3.5.22 Remove Node

```
1 {
2   if (node == null || !contains(node)) {
3     return false;
4   }
}
```

```

5
6     final ArrayList<Node<N, E>> nodes = getNodeList();
7     nodes.remove(node);
8     setNodeList(nodes);
9
10    final ArrayList<Edge<N, E>> edges = getEdgeList();
11    edges.removeAll(node.getFanOut());
12    edges.removeAll(node.getFanIn());
13    setEdgeList(edges);
14
15    for (final Edge<N, E> edge : node.getFanOut()) {
16        edge.removeFromNodes();
17    }
18    for (final Edge<N, E> edge : node.getFanIn()) {
19        edge.removeFromNodes();
20    }
21
22    return true;
23 }

```

3.5.23 Kruskal: Complete

3.5.24 Kuskal: Union Find

3.5.25 Kruskal: Break Condition/Variant

3.5.26 Kruskal: Functionality

3.5.27 Kuskal: Invariant

3.5.28 Prim: Complete

3.5.29 Prim: Break Condition/Variant

3.5.30 Prim: Functionality

Graph:
Kruskal:
Complete

Graph:
Kruskal:
Union Find

Graph:
Kruskal:
Break Condition/Variant

Graph:
Kruskal:
Functionality

Graph:
Kruskal:
Inva-

3.5.31 Prim: Invariante

3.6 Iterativ

3.6.1 Palindrome Check

```
1 {
2     if (s == null) {
3         throw new NullPointerException();
4     }
5     if (StringHelper.isEmpty(s)) {
6         return false;
7     }
8
9     final String lower = s.toLowerCase();
10    for (int i = 0; i < StringHelper.length(lower) / 2; i++) {
11        if (lower.charAt(i) != lower.charAt(s.length() - i - 1)) {
12            return false;
13        }
14    }
15    return true;
16 }
```

Graph:
Prim:
In-
va-
ri-
an-
te

3.6.2 Fibonacci

```
1 {
2     if (n < 0) {
3         throw new IllegalArgumentException();
4     }
5
6     if (n == 0) {
7         return 0;
8     }
9
10    int prev = 0;
11    int result = 1;
12    for (int i = 1; i < n; i++) {
13        result = prev + (prev = result);
14    }
15    return result;
16 }
```

3.6.3 Multiple Palindrome Check

```
1 {
2     if (a == null) {
3         throw new NullPointerException();
4     }
5     if (a.length <= 0) {
6         return null;
7     }
8 }
```

```

9     final boolean[] result = new boolean[a.length];
10    for (int i = 0; i < a.length; i++) {
11        final String s = a[i].toLowerCase();
12        boolean palindrome = true;
13        for (int j = 0; j < StringHelper.length(s) / 2; j++) {
14            if (s.charAt(j) != s.charAt(s.length() - j - 1)) {
15                palindrome = false;
16                break;
17            }
18        }
19        result[i] = palindrome;
20    }
21    return result;
22 }

```

3.7 MIPS

3.7.1 Sort Array

3.7.2 Euklidischer Algorithmus

3.7.3 Pascalsches Dreieck

3.8 Rekursiv

3.8.1 Approximate Square Root

```

1  {
2      if (x < 0 || g < 0 || tolerance < 0) {
3          throw new IllegalArgumentException();
4      }
5
6      if (Math.abs((x / g) - g) < tolerance) {
7          return g;
8      }
9
10     return proxRootRec(x, ((x / g) + g) / 2, tolerance);
11 }

```

3.8.2 Fibonacci

```

1  {
2      if (i <= 0) {
3          return 0;
4      }

```

MIPS:
Sort
Ar-
ray

MIPS:
Eu-
kli-
di-
scher
Al-
go-
rith-
mus

MIPS:
Pas-
cal-
sches
Drei-
eck

```
5     if (i == 1) {
6         return 1;
7     }
8     return fibRec(i - 1) + fibRec(i - 2);
9 }
```

3.9 Single Linked List

3.9.1 Insert First (single)

```
1 {
2     if (el == null || getFirst() == el) {
3         return false;
4     }
5
6     if (getFirst() == null) {
7         setLast(el);
8     }
9     el.setNext(getFirst());
10    setFirst(el);
11    setSize(size() + 1);
12
13    return true;
14 }
```

3.9.2 Insert Last (single)

```
1 {
2     if (el == null || getLast() == el) {
3         return false;
4     }
5
6     if (getFirst() == null) {
7         setFirst(el);
8     } else {
9         getLast().setNext(el);
10    }
11    setLast(el);
12    el.setNext(null);
13
14    setSize(size() + 1);
15
16    return true;
17 }
```

3.9.3 Clone Elements

```
1 {
2     if (el == null) {
3         throw new NullPointerException();
4     }
5
6     final ListElement<T> head = new ListElement<T>(el.getData());
7     ListElement<T> clone = head;
```

```

8     for (ListElement<T> orig = el.next();
9         orig != null; orig = orig.next()) {
10        final ListElement<T> elem = new ListElement<T>(orig.getData());
11        clone.setNext(elem);
12        clone = elem;
13    }
14    return head;
15 }

```

3.9.4 Clone List

```

1 {
2     if (list == null) {
3         throw new NullPointerException();
4     }
5     if (list.isEmpty()) {
6         return new LinkedList<T>();
7     }
8
9     final ListElement<T> head = list.getFirst();
10    final LinkedList<T> result = new LinkedList<T>();
11    ListElement<T> clone = new ListElement<T>(head.getData());
12    result.setFirst(clone);
13    int count = 1;
14    for (ListElement<T> el = head.next(); el != null; el = el.next()) {
15        final ListElement<T> elem = new ListElement<T>(el.getData());
16        clone.setNext(elem);
17        clone = elem;
18        count++;
19    }
20    result.setSize(count);
21    result.setLast(clone);
22    return result;
23 }

```

3.9.5 Duplicate every second Element

```

1 {
2     boolean duplicate = true;
3     for (MListElement<T> el = head; el != null; el = el.getNext()) {
4         if (duplicate) {
5             final MListElement<T> dupl = new MListElement<T>(el.getKey());
6             dupl.setNext(el.getNext());
7             el.setNext(dupl);
8             el = dupl;
9         }
10
11        duplicate = !duplicate;
12    }
13
14    return head;
15 }

```

3.9.6 Get at Index

```

1 {
2     // Dummy call to make Codemonkeys happy.
3     isEmpty();
4
5     int i = 0;
6     for (ListElement<T> el = getFirst(); el != null; el = el.next(), i++) {
7         if (i == idx) {
8             return el;
9         }
10    }
11    return null;
12 }

```

3.9.7 Insert

3.9.8 Insert First

```

1 {
2     if (el == null) {
3         return false;
4     }
5     // Loop detection using Floyd's circle-finding algorithm.
6     boolean run = true;
7     for (ListElement<T> i = el, j = el; run;) {
8         if (i.hasNext()) {
9             i = i.next();
10        } else {
11            run = false;
12            break;
13        }
14        if (j.hasNext() && j.next().hasNext()) {
15            j = j.next().next();
16        } else {
17            run = false;
18            break;
19        }
20
21        if (i == j) {
22            return false;
23        }
24    }
25
26    ListElement<T> last = null;
27    int count = 0;
28    for (ListElement<T> elem = el; elem != null;
29         elem = elem.next(), count++) {
30        if (contains(elem)) {
31            return false;
32        }
33        if (!elem.hasNext()) {
34            last = elem;
35        }
36    }
37    setSize(size() + count);
38    last.setNext(getFirst());
39    setFirst(el);

```

Single
Linked
List:
In-
sert

```

40     if (getLast() == null) {
41         setLast(last);
42     }
43
44     return true;
45 }

```

3.9.9 Insert Last

```

1  {
2      if (el == null) {
3          return false;
4      }
5      // Loop detection using Floyd's circle-finding algorithm.
6      boolean run = true;
7      for (ListElement<T> i = el, j = el; run;) {
8          if (i.hasNext()) {
9              i = i.next();
10         } else {
11             run = false;
12             break;
13         }
14         if (j.hasNext() && j.next().hasNext()) {
15             j = j.next().next();
16         } else {
17             run = false;
18             break;
19         }
20
21         if (i == j) {
22             return false;
23         }
24     }
25
26     ListElement<T> last = null;
27     int count = 0;
28     for (ListElement<T> elem = el;
29         elem != null; elem = elem.next(), count++) {
30         if (contains(elem)) {
31             return false;
32         }
33         if (!elem.hasNext()) {
34             last = elem;
35         }
36     }
37     setSize(size() + count);
38     if (getLast() == null) {
39         setFirst(el);
40     } else {
41         getLast().setNext(el);
42     }
43     setLast(last);
44
45     return true;
46 }

```

3.9.10 Insert (single)

```

1 {
2     if (el == null || idx < 0 || idx > size() || contains(el)) {
3         return false;
4     }
5
6     el.setNext(null);
7
8     if (idx == 0) {
9         el.setNext(getFirst());
10        setFirst(el);
11        if (getLast() == null) {
12            setLast(el);
13        }
14    } else if (idx == size()) {
15        if (getLast() == null) {
16            setFirst(el);
17            setLast(el);
18        } else {
19            getLast().setNext(el);
20            setLast(el);
21        }
22    } else {
23        int i = 1;
24        for (ListElement<T> elem = getFirst();
25            elem != null; elem = elem.next(), i++) {
26            if (i == idx) {
27                el.setNext(elem.next());
28                elem.setNext(el);
29                break;
30            }
31        }
32    }
33    setSize(size() + 1);
34    return true;
35 }

```

3.9.11 Invert

```

1 {
2     if (head == null) {
3         return null;
4     }
5
6     ListElement<T> next = null;
7     ListElement<T> cur = head;
8     for (ListElement<T> el = head.next(); el != null; el = next) {
9         next = el.next();
10
11        el.setNext(cur);
12        cur = el;
13    }
14    head.setNext(null);
15    return cur;
16 }

```

3.9.12 Merge Linked Lists

```

1  {
2      if (left == null || right == null || comp == null) {
3          throw new IllegalArgumentException();
4      }
5
6      MListElement<T> result = null;
7      for (MListElement<T> i = left, j = right, merged = null;
8          i != null || j != null; ) {
9          final MListElement<T> use;
10         if (i == null) {
11             use = j;
12             j = j.getNext();
13         } else if (j == null) {
14             use = i;
15             i = i.getNext();
16         } else if (comp.compare(i.getKey(), j.getKey()) < 0) {
17             use = i;
18             i = i.getNext();
19         } else {
20             use = j;
21             j = j.getNext();
22         }
23
24         if (merged != null) {
25             merged.setNext(use);
26         }
27         merged = use;
28         if (result == null) {
29             result = merged;
30         }
31     }
32
33     return result;
34 }

```

3.9.13 Remove

3.9.14 Remove First

```

1  {
2      final ListElement<T> first = getFirst();
3      if (first != null) {
4          setFirst(first.next());
5          setSize(size() - 1);
6          if (size() == 0) {
7              setLast(null);
8          }
9      }
10     return first;
11 }

```

Single
Lin-
ked
List:
Re-
mo-
ve

3.9.15 Remove Last

```

1 {
2     if (getFirst() == null) {
3         return null;
4     }
5
6     final ListElement<T> result = getLast();
7     if (getFirst() == getLast()) {
8         setFirst(null);
9         setLast(null);
10    } else {
11        ListElement<T> secondLast = getFirst();
12        while (secondLast.next() != getLast()) {
13            secondLast = secondLast.next();
14        }
15
16        secondLast.setNext(null);
17        setLast(secondLast);
18    }
19    setSize(size() - 1);
20    return result;
21 }

```

3.9.16 Remove duplicated Elements

```

1 {
2     if (head == null) {
3         return null;
4     }
5
6     ListElement<T> prev = head;
7     for (ListElement<T> el = head.next(); el != null; el = el.next()) {
8         if (comp.compare(prev.getData(), el.getData()) == 0) {
9             prev.setNext(el.next());
10        } else {
11            prev = el;
12        }
13    }
14    return head;
15 }

```

3.10 String Operations

3.10.1 Prefix Check

```

1 {
2     if (a == null || b == null) {
3         return false;
4     }
5
6     final String lowerA = StringHelper.toLowerCase(a);
7     final String lowerB = StringHelper.toLowerCase(b);
8     for (int i = 0; i < lowerA.length(); i++) {
9         if (i >= lowerB.length() || lowerA.charAt(i) != lowerB.charAt(i)) {
10            return false;
11        }
12    }

```

```
13     return true;
14 }
```

3.10.2 Simple String Matcher

```
1  {
2      if (S == null || T == null) {
3          throw new IllegalArgumentException();
4      }
5
6      final String haystack = S.toLowerCase();
7      final String needle = T.toLowerCase();
8
9      final ArrayList<int[]> tupels = new ArrayList<int[]>();
10     final ArrayList<Integer> result = new ArrayList<Integer>();
11     for (int i = 0; i < haystack.length(); i++) {
12         tupels.add(new int[] { i + 1, -1 } );
13
14         final java.util.Iterator<int[]> it = tupels.iterator();
15         while (it.hasNext()) {
16             final int[] tuple = it.next();
17             tuple[1] += 1;
18             if (haystack.charAt(i) != needle.charAt(tuple[1])) {
19                 it.remove();
20             } else if (tuple[1] == needle.length() - 1) {
21                 it.remove();
22                 result.add(tuple[0]);
23             }
24         }
25     }
26     return result;
27 }
```

3.11 Lambda

3.11.1 Lambda Expressions/Strategy Pattern

doArrayWork

```
1  {
2      final Integer[] result = new Integer[array.length];
3      for (int i = 0; i < array.length; i++) {
4          result[i] = array[i];
5      }
6      for (int i = 0; i < ops.length; i++) {
7          getOperations()[ops[i]].doSomethingOnArrays(result);
8      }
9      return result;
10 }
```

makeOperations

```
1  {
2      getOperations()[0] = arr -> {
3          for (int i = 0; i < arr.length; i++) {
4              arr[i] = arr[i] * arr[i];
5          }
6      }
7  }
```

```
6     return arr;
7 };
8 getOperations()[1] = arr -> {
9     for (int i = 0; i < arr.length; i++) {
10        arr[i] = arr[i] * 2;
11    }
12    return arr;
13 };
14 getOperations()[2] = arr -> {
15     for (int i = 0; i < arr.length; i++) {
16        arr[i] = arr[i] + 2;
17    }
18    return arr;
19 };
20 }
```