Opgaveløsninger
(sæt 6)

Opgave 20: 8.2 (1 point)

Stabile metoder:
- insertion sort
- mergesort

Ikke-stabile metoder:
- Shellsort
- quicksort

Både insertion sort og mergesort er stabile, så længe sammenligninger for lighed ikke ødelægger ordenen. Implementeringen af mergesort i lærebogen på side 297 er imidlertid *ikke* stabil. Stabilitet kan opnås ved at erstatte '<' i linje 18 med '<='.

Det er let at se, at hverken Shellsort eller quicksort er stabil. Når et af to ens elementer sammenlignes med et tredje, kan ordenen af de to ens elementer ændres.

Opgave 21: 8.11 (2 point)

Antal sammenligninger (ifølge sætningen på side 314 i lærebogen) ≥ log(4!) = log(24) = 4.58.

Således må antal sammenligninger være mindst 5.

For at forenkle besvarelsen af opgaven indføres en metode, `comparator`, der sammenligner to elementer i et array og ombytter dem, hvis det første er større end det andet.

```
void comparator(int i, int j) {
    if (a[i].compareTo(a[j]) > 0)
        swap(a, i, j);
}
```

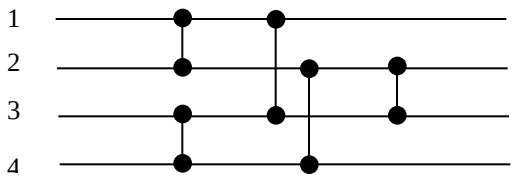
En måde at sortere de 4 elementer er da følgende:

```
comparator(1,2);
comparator(2,3);
comparator(3,4);        // nu er a[4] maksimum af de 4 elementer
comparator(1,2);
compatator(2,3);        // nu er a[3] maksimum af de resterende 3
compatator(1,2);        // nu er a[2] maksimum af de resterende 2,
                        // og dermed er a[1] det mindste af de 4 elementer
```

Her bruges 6 sammenligninger. Imidlertid kan man ved at bruge de små grå finde frem til en løsning, der bruger 1 sammenligning mindre.

```
comparator(1,2);
comparator(3,4);
comparator(1,3);        // nu er a[1] minimum af de 4 elementer
comparator(2,4);        // nu er a[4] maksimum af de 4 elementer
comparator(2,3);        // nu er a[2] og a[3] ordnet,
                        // og dermed er alle 4 elementer sorteret
```

Algoritmen kan anskueliggøres ved et såkaldt sorteringsnetværk:



Heraf fremgår, hvilke operationer, der kan udføres i parallel.

Opgave 22: OneWayList (4 – 6 point)

(a)

```
public void addFirst(Object obj) {
    header.next = new Node(obj, header.next);
}

public boolean isEmpty() {
    return header.next == null;
}
```

(b)

```
private class OneWayListIterator implements Iterator {
    OneWayListIterator() {
        current = header;
    }

    public boolean hasNext() {
        return current.next != null;
    }

    public Object next() {
        if (current.next == null)
            throw new NoSuchElementException();
        current = current.next;
        return current.data;
    }

    public void remove() {
        throw new UnsupportedOperationException();
    }

    private Node current;
}
```

(c)

```
public void reverse() {
    Node prev = null, current = header.next;
    while (current != null) {
        Node next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    header.next = prev;
}
```

(d)

En implementering, der anvender sortering ved indsættelse:

```
public void sort() {
    selectionSort();
}

private void selectionSort() {
    for (Node last = header; last.next != null; last = last.next) {
        Node prev = last, current, minPrev = last, min = null;
        while ((current = prev.next) != null) {
            if (min == null ||
                (comparator != null ?
                 comparator.compare(current.data, min.data) < 0 :
                 ((Comparable) current.data).compareTo(min.data) < 0)) {
                minPrev = prev;
                min = current;
            }
            prev = current;
        }
        minPrev.next = min.next;
        min.next = last.next;
        last.next = min;
    }
}

private Comparator comparator;

public void sort(Comparator comp) {
    comparator = comp;
    sort();
    comparator = null;
}
```

En implementering, der anvender sortering ved fletning:

```
public void sort() {
    header.next = mergeSort(header.next);
}

private Node mergeSort(Node n) {
    if (n == null || n.next == null)
        return n;
    Node a = n, b = n.next;
    while (b != null && b.next != null) {
        n = n.next;
        b = b.next.next;
    }
    b = n.next;
    n.next = null;
    return merge(mergeSort(a), mergeSort(b));
}

private Node merge(Node a, Node b) {
    Node header = new Node(null, null);
    Node c = header;
    while (a != null && b != null) {
        if (comparator != null ?
            comparator.compare(a.data, b.data) < 0 :
            ((Comparable) a.data).compareTo(b.data) < 0) {
            c.next = a;
            c = a;
            a = a.next;
        } else {
            c.next = b;
            c = b;
            b = b.next;
        }
    }
    c.next = a == null ? b : a;
    return header.next;
}
```