Opgaveløsninger
(sæt 3)

Opgave 9: 5.6 (2 point)

a. `low < high`

Når `low` bliver lig med `high`, forlades løkken straks. Derfor undersøges det ikke, om elementet `a[low]` er lig med søgenøglen. En sammenligning med `a[low]` udføres aldrig. Dermed kan vi risikere, at nøglen findes i arrayet, uden at det opdages.

b. `mid = low + high / 2`

Antag at `low` bliver lig med `high`, og at `low` på dette tidspunkt er større end `2`. Så vil `mid` blive sat til en værdi, der er større end `low`, og dermed også større end `high`. I heldigste fald får vi en køretidsfejl (`IndexOutOfBoundsException`). I værste fald terminerer løkken aldrig.

c. `low = mid`

Antag at `high` bliver lig med `low`. Så vil sætningen `mid = (low + high) / 2` bevirke, at `mid` bliver sat lig med `low`. Hvis nu `a[mid].compareTo(x) < 0`, så sættes `low` til `mid`. Men da `mid` jo er lig med `low`, ændres `low` ikke. Således bliver `low` ved med at være lig med `high`, og løkken terminerer aldrig.

d. `high = mid`

Antag at `high` bliver lig med `low`. Så vil sætningen `mid = (low + high) / 2` bevirke, at `mid` bliver sat lig med `high`. Hvis nu `a[mid].compareTo(x) > 0`, så sættes `high` til `mid`. Men da `mid` jo er lig med `high`, ændres `high` ikke. Således bliver `high` ved med at være lig med `low`, og løkken terminerer aldrig.

Opgave 10: 5.15, spørgsmål a (2 point)

#1. O(n)	En simpel løkke af orden n.
#2. O(n)	En simpel løkke af orden n (cirka n/2 iterationer)
#3. O(n^2)	To indlejrede løkker, hver af orden n.
#4. O(n)	To konsekutive løkker, hver af orden n .
#5. O(n^3)	To indlejrede løkker af orden n og n^2.
#6. O(n^2)	To indlejrede løkker hver af potentiel orden n.
#7. O(n^5)	Tre indlejrede løkker af orden n, n^2 og n^2.
#8. O(log n)	Efter fordoblingsprincippet uføres løkken cirka log n gange. Der trykfejl i bogen. Variablen `i` skal initialiseres til 1.

Opgave 11: 5.24 (2 point, ikke-obligatorisk)

En effektiv algoritme kan konstrueres ud fra nedenstående ide (problemudtynding):

Ide: Hvis to elementer er forskellige, og de pågældende elementer fjernes, så vil en eventuel vinder for de oprindelige elementer også være en vinder for de resterende.

Bemærk at det modsatte er ikke tilfældet. F.eks. har listen (1, 2, 5, 5, 3) ingen vinder, men hvis 1 og 2 fjernes, så bliver 5 ny vinder.

Ideen udnyttes i følgende algoritme:

Algoritme: Elementerne gennemløbes i rækkefølge. Hver gang to er forskellige, eliminerer vi dem begge. Vinderen i den resterende liste (kandidaten) bestemmes, hvorefter det undersøges, om denne er vinder i den originale liste.

Men hvad gør vi, hvis to successive stemmer er ens?

Svaret er, at vi blot skal vedligeholde to variable:

> `candidate`, der udpeger en potentiel vinder, og
> `count`, der angiver hvor mange gange `candidate` ikke har kunnet parres med en anden kandidat.

Derefter undersøges ved simpel tælling, om `candidate` er vinder.

Vi når dermed frem til en Java-metode som den, der er vist nedenfor.

```
Object majority(Object a[]) {
    int count = 0;
    Object candidate = null;
    // Find a candidate
    for (i = 0; i < a.length; i++)
        if (count == 0) {
            candidate = a[i];
            count = 1;
        } else if (a[i].equals(candidate))
            count++;
        else
            count--;
    // Test if candidate is a winner
    count = 0;
    for (i = 0; i < a.length; i++)
        if (a[i].equals(candidate))
            if (++count > a.length/2)
                return candidate;
    return null;
}
```

Det er let at se, at algoritmens køretid er O(N).