# Opgaver uge 10

## Formål med opgaverne

Efter disse øvelser skal du kunne måle den faktiske køretid af nogle sorteringsrutiner på små og store datasæt. Du skal også kunne implementere (simple) søgealgoritmer i Java og redegøre for deres køretider, samt anvende et interface til at generalisere en sorteringsmetode.

## Opgave 1

Lav følgende opgaver fra noterne *Searching and sorting with Java* som kan findes fra kursets forelæsningsplan: Opgave 4.10.5 (køretid for udvalgssortering), 5.8.5 (køretid for quicksort), 6.7.5 (køretid for hobsortering). Hver opgave går ud på at måle køretiden for sortering af et array med `n` tilfældige tal. Du kan evt. gemme 6.7.5. til næste uge, når vi har gennemgået hobsortering til forelæsningerne.

Vink 1: Du kan bruge nedenstående simple `Timer`-klasse til at måle hvor lang tid udførelsen af en sorteringsrutine tager. Læg mærke til at den måler sandtid ("wall clock time") så målingen bliver forkert hvis der kører en masse andre programmer (browsere, tekstbehandling) der bruger cpu-tid på maskinen samtidig.

```
public class Timer {
  private long start;
  public Timer() {
    start = System.currentTimeMillis();
  }
  public double check() {
    return (System.currentTimeMillis()-start)/1000.0;
  }
}
```

Ovenstående `Timer`-klasse kan for eksempel bruges sådan her:

```
Timer t = new Timer();
selsort(arr, n);
System.out.println(t.check() + " sekunder");
```

Vink 2: Lav målingen for `n` lig med 1 000, 2 000, 4 000, 8 000, 16 000, osv indtil køretiden overstiger fx 2 sekunder. Det kan du automatisere ved passende anvendelse af en `while`-løkke.

## Afleveringsopgave G5

Lav opgave 7.4.1 fra noterne *Searching and sorting with Java* som kan findes fra kursets forelæsningsplan.

Giv et estimat af bedste og værste køretid for boblesortering af $n$ tal, Java-implementation af boblesortering, og måling af køretid i samme stil som opgave 1.

## Opgave 3

Denne opgave generaliserer sorteringsalgoritmerne fra noterne til at kunne sortere generelle klasser med ordninger. Målet med opgaven er at kunne anvende den samme sorteringsmetode til at sortere et array af bøger først alfabetisk og derefter efter udgivelsesår. Den illustrerer også begrænsningerne ved teknikerne angivet i afsnit 8 i noterne *Searching and sorting with Java*. Læs afsnit 8 før du går igang.

(i) Udvid `searching-sorting` projektet med et interface `Ordered` der blot har en operation `less` som angivet i noterne *Searching and sorting with Java* afsnit 8.1, og udvid også med klassen `OrderedString` som i afsnit 8.3.

(ii) Tilpas quicksort metoden, så den kan sortere arrays af elementer af interface `Ordered`.

(ii) Lav en klasse `Book` som repræsenterer bøger. En bog har en forfatter, en titel og et udgivelsesår. De første bør implementeres som felter af klassen `OrderedString` og det sidste som et felt af typen `int`. Implementer også en `toString()` metode, som returnerer en streng med disse data.

(iii) Lav en subklasse `BookAlphabetical` af `Book` som implementerer interface `Ordered`. Metoden `less` skal sammenligne to bøger alfabetisk, først efter forfatter, derefter titel. Det kan være nyttigt at have en konstruktor i denne klasse, der tager et objekt af klassen `Book` som parameter.

(iv) Lav en klasse `Tester` til at teste quicksort og `BookAlphabetical`. Denne klasse bør have en privat metode som tager et array af `Book` og returnerer et array af `Book`, som indeholder elementerne sorteret alfabetisk.

Vink: Du kan ikke kalde `quicksort` på et array af `Book`; du skal bruge et array af `BookAlphabetical`. For at lave sådan et, er du nødt til at konstruere et nyt array ved at løbe det givne array igennem og konvertere hvert enkelt element til et i subklassen. Det er her det er praktisk at have en konstruktor der tager et element af typen `Book`.

(v) Lav en subklasse `BookByYear` af `Book` som implementerer interface `Ordered`. Metoden `less` skal sammenligne to bøger efter udgivelsesår. Hvis to bøger har samme udgivelsesår sammenlignes de alfabetisk, først efter forfatter, derefter titel. Ligesom med `BookAlphabetical` kan det være nyttigt at have en konstruktor i denne klasse, der tager et objekt af klassen `Book` som parameter.

(vi) Tilføj en privat metode til `Tester` som tager et array af `Book` og returnerer et array af `Book`, som indeholder elementerne sorteret efter årstal.

(vii) Tilføj en metode `test()` i klassen `Tester`, der konstruerer et array af 5-10 bøger og skriver det ud først sorteret alfabetisk, og derefter sorteret efter udgivelsesår.

**Aflever:** Udskrift af den opdaterede quicksort metode, samt alle nye klasser du har defineret.

## Opgave 4

Dette er en ekstra opgave for dem der har brug for lidt ekstra udfordring. Du er naturligvis stadig velkommen til at aflevere din besvarelse til instruktorerne.

Formålet med opgaven er at implementere en dobbelt-hægtet liste (engelsk: doubly-linked list) af heltal. Dobbelthægtede lister er implementeret i Java API som klassen `LinkedList`. Du kan finde yderligere information om dobbelt-hægtede lister på wikipedia (f.eks. er tegningen der nok nyttig).

(i) Lav en klasse `Node`, som repræsenterer enkelte celler i listen. Klassen `Node` skal have tre private felter: Et kaldet `value` af typen `int`, og to kaldet `next` og `previous` af typen `Node`. Lav en konstruktor der tager et heltal og initialiserer `value` med dette og lader de to andre felter være `null`. Lav også en metode der returnerer `value` og accessor metoder for `next` og `previous` og metoder til at sætte værdien af disse.

(ii) Lav en klasse `MyLinkedList` som repræsenterer dobbelthægtede lister. Denne klasse skal have to private felter `first` og `last` af typen `Node`. Konstruktoren skal sætte disse til `null`.

Idéen med en dobbelthægtet liste er at et objekt repræsenterer listen af heltal, der forekommer ved at læse værdien af den `Node`, der står i `first` feltet, og derefter følge `next` pointeren til den næste `Node` og så fremdeles. Man kan læse listen baglæns ved at starte med feltet `last` og følge `previous` pointerne.

(iii) Lav en metode `add` i `MyLinkedList` som tilføjer et element sidst i listen. Metoden skal tage et heltal som parameter og oprette en instans `newNode` af klasse `Node` med dette heltal som værdi. Hvis listen er tom, dvs. hvis

`first` er `null`, så skal både `first` og `last` sættes til `newNode`. Hvis listen ikke er tom skal den `Node` der peges på fra `last` (kald denne `lastNode`) have sat sin `next` værdi til `newNode`, `newNode` skal have sat sin `previous` til `lastNode` og i listen skal `last` sættes til `newNode`.

(iv) Lav en metode `get` i `MyLinkedList` som tager et heltal `i` som input og returnerer det `i`'te element i listen. Returtypen for `get` skal være `int`.

(v) Test `get` og `add` ved at lave en metode i en testerklasse som opretter en liste på f.eks 5 elementer (ved at kalde `add` 5 gange) og læser værdien af det tredie element ved at kalde `get(2)`.

(vi) Hvordan afhænger køretiderne for `add` og `get` af længden af listen? Kan du optimere `get` så den bliver hurtigere til at læse elementerne i den sidste halvdel af listen?

(vii) Kig i dokumentationen for `LinkedList` klassen. Implementer andre metoder fra `LinkedList` i din `MyLinkedList` klasse, f.eks. `size`, `set` og `remove(int index)`.

**Aflever:** Udskrift af de klasserne `Node` og `MyLinkedList` samt tester klassen, og dine kommentarer om køretider.