# DAT102 – Våren 2021

Øving 6
Utlevert 1. mars

Del av obligatorisk 3

## Oppgave 1

Når man skal implementere en algoritme, kan dette gjøres på flere måter. Ofte vil det være mulig å gøre «triks» som sparer tid. Algoritmen vil ha samme orden, men konstanten i leddet som vokser raskest blir mindre. Dere skal undersøke om slike triks har betydning i sorterting ved innsetting (insertion sort).

Starten på den indre løkken kan se slik ut

```
while (!ferdig && j >= 0)
```

der første del av betingelsen er at vi ikke har funnet rett plass for elementet som skal settes inn og andre del at det er flere elementer å sammenligne med. Dersom vi flytter minste elementet fremst før vi starter selve sorteringen, kan betingelsen forenkles siden vi aldri skal sette inn et element i posisjon 0 i tabellen. En av fordelene til sortering ved innsetting er at den er stabil (stable). Det vil si at like elementer vil ha samme innbyrdes rekkefølge etter sortering. For å beholde denne egenskapen kan vi gå fra høyre mot venstre i tabellen og bytte om naboelementer om de står feil i forhold til hverandre. Da er minste elementet kommet først.

a) Modifiser koden om angitt ovenfor og se om det har betydning for tidsbruken. La antall elementer være så stort at det tar minst 5 sekunder å utføre sorteringen. Skriv kort hva dere observerer. For å generere tilfeldige data og måle tid, se til slutt i øvingen.

b) Modifiser koden slik at i stedet for å sette inn ett element om gangen, setter vi inn to. Så lenge det største elementet er mindre enn elementet vi sammenligner med i sortert del, så kan vi flytte elementet to plasser til høyre. Når vi finner rett plass for største, forsetter vi som vanlig med å sette inn det minste. Kombiner med å flytte minste elementet først (som i a) før sorteringen starter. Pass på at metoden fungerer for både odde og jevne n. Skriv kort hva dere observerer.

I tillegg til observasjonene, skal dere levere kode for både a) og b).

## Oppgave 2

I denne oppgaven skal dere få praktisk erfaring med hvor lang tid sorteringsmetodene trenger for å sortere tabeller med heltall. Det blir forskjell (men bare i konstanten framfor leddet som vokser raskest) for en og samme sorteringsmetode om vi for velger primitiv type heltall (int) eller om vi velger heltalsobjekt (Integer). Implementer metodene nedenfor i Java.

- Sortering ved innsetting (Insertion sort)
- Utvalgssortering (Selection sort)
  Samme som Plukksortering som vi så i DAT100
- Kvikksortering (Quick sort)
- Flettesortering (Merge sort)

Ta gjerne utgangspunkt i en tabell av heltalsobjekt (*Integer*). Før dere går videre, kontroller at sorteringsmetodene er korrekte ved å bruke de på en liten tabell (n = 10) og skriv ut tabellen etter sortering.

La T(n) være tiden det tar å sortere n element. At en algoritme bruker tid O(f(n)) betyr at T(n) = c*f(n). I sorteringsmetodene ovenfor er f(n) lik $n^2$ eller $n*\log_2 n$. Bestem først c slik at T(n) = c*f(n). Dette kan gjøres ved å måle tiden for en spesiell n-verdi, for eksempel n = 32000, og så løse ligningene med c som ukjent. Konstanten c er avhengig av

- algoritmen
- implementasjonen
- maskinen som kjører programmet

Til slutt i oppgaven finner du forklaring på hvordan du kan måle tiden og hvordan du kan få en tabell med tilfeldige tal.

a) For hver sorteringsmetode kan utskriften se ut som under (men med overskrift og f(n) erstattet med det som er relevant). Diskuter hvordan de teoretiske resultatene samsvarer med de målte og prøv å forklare eventuelle avvik.

**Resultat Kvikksortering** (tilsvarande tabeller for de andre metodene)**:**

| **n** | **Antall målinger** | **Målt tid (gjennomsnitt)** | **Teoretisk tid c*f(n)** |
|---|---|---|---|
| 32000 | | | |
| 64000 | | | |
| 128000 | | | |

På første linje i tabellen vil målt og teoretisk tid være like dersom dere bruker 32000 for å bestemme c.

b) Prøv å sortere en tabell der alle elementene er like med Quicksort og mål tiden. Forklar hva som skjer og hvorfor det skjer?

## Hvordan måle utføringstider i Java

For å måle brukt tid kan du bruke klassene **`Instant`** og **`Duration`**. Generelt om pakken java.time:

https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html.

Det er også mulig å bruke: **`System.nanoTime().`** Se

https://www.baeldung.com/java-measure-elapsed-time

Av ulike grunner blir ikke tidtakingen helt nøyaktig, spesielt for små tider. Derfor kan det for små n-verdier være aktuelt å utføre den kritiske delen flere ganger og så bruke gjennomsnittstiden. Dette gjør dere ved å legge den kritiske koden innenfor en løkke, utføre løkken et visst antall ganger og så finne gjennomsnittlig tid. Eksempel på kode:

```
Random tilfeldig = new Random(…);
int n = 32000;
int antal = 10;

Integer[][] a = new Integer[antal][n];

// set inn tilfeldige heiltal i alle rekker
for (int i = 0; i < antal; i++){
      for (int j = 0; j < n; j++){
            a[i][j] = tilfeldig.nextInt();
      }
}

// start tidsmåling
for (int i = 0; i < antal; i++){
      sorter(a[i]); // blir ein eindimensjonal tabell
}
// slutt tidsmåling
```

## Hvordan få en tabell med n tilfeldige heltall

Java har en forhåndsdefinert klasse for tilfeldige tal som heter Random. Den har to konstruktører

- `Random()` - bruker systemklokken for å gi generatoren en startverdi.
- `Random(long starverdi)` - vi gir startverdi. Det vil si at vi kan generere nøyaktig same tallrekke på nytt flere ganger. Dette er aktuelt ved testing.

Etter å ha initiert generatoren, kan du bruke flere metoder, men den vi trenger i øvelsen er:

- `int nextInt()` - som gir et tilfeldig heltall. Skisse for å lage in tabell med tilfeldige heltall er vist under:

```
import java.util.Random;
...
Random tilfeldig = new Random(); // brukar maskinen sin klokke for å gi startverdi
...
for (int i=0; i < n; i++){
   tabell[i] = tilfeldig.nextInt();
}
```

Dersom dere vil ha heltall fra og med 0 til (men ikke med) M, kan dere skrive

```
tabell[i] =  tilfeldig.nextInt(M);
```