Algoritmer og datastrukturer
Kapittel 1 - Delkapittel 1.1

1.1 Algoritmer og effektivitet

1.1.1 Hva er en algoritme?

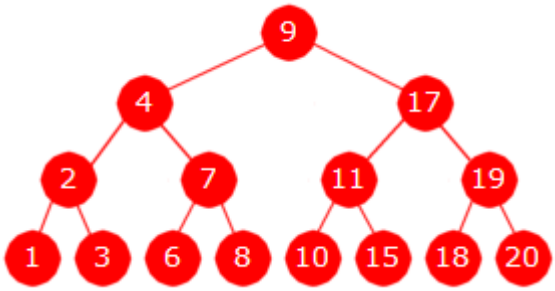
al-Khwarizmi

En *algoritme* (eng: algorithm) er en beskrivelse av de operasjonene som skal til for å løse en gitt oppgave - dvs. en oppskrift. Ordet algoritme kommer fra navnet til matematikeren og astronomen *al-Khwarizmi* (ca. 780 - 850). Hans bok *Al-jabr wa'l muqabalah* handler om det vi i dag kaller *algebra*. Den inneholder bl.a. en oppskrift på å løse 2. gradsligninger. Ordet algebra kommer av ordet «al-jabr» – det første ordet i bokens tittel. Ordet algoritme har oppstått slik: *al-Khwarizmi -> algorismi -> algoritme*. I datavitenskap (eng: computer science) bruker man en mer formell definisjon av begrepet algoritme. På engelsk kan det f.eks. hete:

> *An algorithm is a well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time.*

En *datastruktur* (eng: data structure) er den måten en samling dataverdier er organisert. Vårt fag heter *Algoritmer og datastrukturer* fordi våre algoritmer vanligvis er nøye knyttet til en datastruktur. Hvis vi f.eks. skal lete etter en bestemt verdi i en samling verdier, er det lett å forstå at letemetoden vil være avhengig av hvordan verdiene er organisert. Har verdiene en bestemt «orden»? Eller ligger de tilfeldig fordelt? Vi vil se at jo mer «orden» det er, jo mer effektive letemetoder kan vi normalt lage.

Tegningen til høyre viser et *binært søketre* (eng: binary search tree) med 15 verdier. Det er en datastruktur der leting etter verdier vil kunne skje på en effektiv måte. Under er det en tabell (eng: array) der de samme 15 verdiene ligger uten noen bestemt orden. Dette kaller vi en uordnet eller usortert tabell. Der er det ikke mulig å lete på en effektiv måte.



Figur 1.1.1 a): Et binært søketre med 15 verdier

8	4	17	10	6	20	1	11	15	3	18	9	2	7	19

Figur 1.1.1 b): En (uordnet) tabell med 15 verdier

Oppgaver til Avsnitt 1.1.1

1. Søk på internett med navnet *al-Khwarizmi* som søkeord. Hva finner du?
2. Lag en norsk versjon av den engelske algoritme-definisjonen over!
3. Søk på internett etter alle sider som inneholder ordene *data structure* (eventuelt *data structures)* og *algorithm* (eventuelt *algorithms)*. Hvor mange treff får du?

1.1.2 Eksempel: Den største verdien i en tabell

En ikke uvanlig oppgave i databehandling er å finne hvor den største verdien i en tabell ligger. Tabellen kan f.eks. inneholde en tilfeldig samling heltall slik som i *Figur* 1.1.1 *b)*. Dvs. datastrukturen er en uordnet heltallstabell. Den må inneholde minst ett tall for at problemet skal ha mening. I Java er det lett å lage en algoritme/metode for dette:

```
public static int maks(int[] a)  // a er en heltallstabell
{
  if (a.length < 1)
    throw new java.util.NoSuchElementException("Tabellen a er tom!");

  int m = 0;  // indeks til foreløpig største verdi

  for (int i = 1; i < a.length; i++) // obs: starter med i = 1
  {
    if (a[i] > a[m]) m = i;  // indeksen oppdateres
  }

  return m;  // returnerer indeksen/posisjonen til største verdi

} // maks
```

***Programkode* 1.1.2**

Programkode 1.1.2 starter med at tabellens lengde sjekkes. Hvis den har lengde lik 0, er tabellen tom. En tom tabell har ingen største verdi. Det gjøres ingen eksplisitt sjekk på om tabellen eksisterer, dvs. om *a* er *null* eller ikke. Kompilatoren setter inn kode for det.

Hele tabellen må gjennomsøkes før vi kan avgjøre hvor den største ligger. Heltallsvariablen *m* i *Programkode* 1.1.2 vil til slutt inneholde indeksen/posisjonen til tabellens største verdi. Legg merke til at hvis det er flere forekomster av den største verdien, vil det være posisjonen til den første av dem fra venstre som returneres.

I tabellen på tegningen under (*Figur* 1.1.2) er posisjonen (eller indeksen) til hvert tabellelement satt opp. Hvis tabellen heter *a*, vil metodekallet *maks*(*a*) returnere posisjonen 5 siden det største tallet (dvs. 20) ligger i posisjon 5. Husk at 0 er første posisjon i en tabell, dvs. at *a*[0] er tabellens første verdi!!

8	4	17	10	6	20	1	11	15	3	18	9	2	7	19
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Figur 1.1.2 : Den største verdien ligger i posisjon 5

Oppgaver til Avsnitt 1.1.2

1. Anta at vi har en *min*-metode, dvs. en metode som finner (og returnerer) posisjonen til den minste verdien i en tabell. Hva ville metodekallet *min*(*a*) returnere med hvis *a* er tabellen i *Figur* 1.1.2?
2. Lag en *min*-metode på samme måte som *maks*-metoden i *Programkode* 1.1.2, dvs. en metode som finner (og returnerer) posisjonen til den minste verdien i en tabell.
3. Hvis den største verdien forekommer flere ganger, vil *maks*-metoden returnere posisjonen til den første av dem. Hva må endres for at den skal returnere posisjonen til den siste?

1.1.3 Algoritmers effektivitet

En algoritme er som nevnt en beskrivelse av de arbeidsoperasjonene som skal til for å løse en bestemt oppgave. De oppgavene vi skal studere vil være slike som kan løses ved hjelp et program (eller kanskje en metode) skrevet i et programmeringsspråk. Vi bør derfor skille mellom 1) *selve oppgaven*, 2) *en algoritme for å løse oppgaven* og 3) *en implementasjon av algoritmen*. Poenget er at det ofte vil være flere mulige måter å løse en oppgave på, dvs. flere mulige algoritmer. Videre vil en bestemt algoritme kunne implementeres på flere måter. Å implementere betyr å «oversette» til programkode.

Grekerne laget matematiske algoritmer

Det første kravet vi må stille til en algoritme er at den er korrekt, dvs. at den løser oppgaven. En algoritme som ikke virker eller som gir oss galt resultat, har ingen verdi. Men det er også gunstig at vi bruker den algoritmen som er mest *effektiv* (hvis en slik finnes) til å løse oppgaven. Den skal brukes i dataprogrammer. Vi kunne derfor si at den er effektiv hvis datamaskinen bruker kort tid på å eksekvere (utføre) den. En slik definisjon på effektivitet er imidlertid problematisk fordi tidsforbruket er avhengig av en serie ytre faktorer, dvs. faktorer som ligger utenfor algoritmen. Vi vet at jo kraftigere datamaskinen er, jo korterer tid vil programmet normalt bruke. Også faktorer som programmeringsspråk og kompilator er med på å bestemme tidsforbruket. F.eks. vil en algoritme implementert i C/C++ i en del tilfeller bli eksekvert raskere enn en tilsvarende implementering i Java.

En algoritmes tidsforbruk vil normalt være avhengig av de dataverdiene som algoritmen arbeider med. Hvis en bruker tidsforbruket til å si noe om effektiviteten, må en gjøre mange testkjøringer med datasett som er forskjellige både i innhold og størrelse. Målinger av tidsforbruk er imidlertid godt egnet til å sammenligne forskjellige algoritmer (og forskjellige implementasjoner) for samme oppgave. Det bør imidlertid være signifikant forskjell i tidsforbruket før vi kan si at en algoritme (eller en implementasjon) er mer effektiv enn en annen.

Vi kan måle algoritmens tidsforbruk.

En mer objektiv (dvs. uavhengig av ytre faktorer) måte å vurdere en algoritmes effektivitet på er å se på de *arbeidsoperasjonene* som utføres i algoritmen. Da kunne vi si at en algoritme er effektiv hvis den relativt sett (dvs. med hensyn på oppgavens størrelse) utfører «få arbeidsoperasjoner».

Hva menes så med en *arbeidsoperasjon*? En arbeidsoperasjon er enten *sammensatt* (eng: compound) eller *grunnleggende* (eng: primitive). Den kalles grunnleggende hvis den ikke kan deles opp i enklere operasjoner. Det er ingen bestemt regel for dette, men en kunne for eksempel (på et mer overordnet nivå) kalle flg. operasjoner for grunnleggende:

- en *tilordning* - en variabel får en verdi
- en *tabelloperasjon* - en tabellverdi aksesseres
- en *sammenligning* - to verdier sammenlignes
- en *regneoperasjon* - f.eks. en addisjon av to tall

En operasjon er sammensatt hvis den består av flere grunnlegggende operasjoner. F.eks. er `if (a[i] > a[m])` sammensatt av to tabelloperasjoner og én sammenligning.

Vi kan finne en formel for en algoritmes effektivitet ved å telle opp antallet ganger det utføres en grunnleggende operasjon. Ta *maks*-metoden i *Programkode* 1.1.2 som eksempel. Anta at tabellen a har n verdier, dvs. $a.length$ er lik n. Vi får flg. regnskap:

- Det utføres én sammenligning for å avgjøre om tabellen har innhold: (1)
- Det opprettes en hjelpevariabel m som får 0 som startverdi: (1)
- Løkkevariabelen i opprettes og får 1 som startverdi: (1)
- I *for*-løkken utføres sammenligningen $i < a.length$ n ganger: (n)
- I *for*-løkken utføres addisjonen $i{+}{+}$ $n - 1$ ganger: ($n - 1$)
- Setningen $if\ (a[i] > a[m])$ utføres $n - 1$ ganger. Den består av to tabelloperasjoner og én sammenligning: $3(n - 1)$
- Tilordningen $m = i$ utføres hver gang $a[i] > a[m]$ er sann: (x)
- Verdien til m returneres: (1)

Sum: $1 + 1 + 1 + n + n - 1 + 3(n - 1) + x + 1 = 5n + x$

Verdien til x er antall ganger sammenligningen $a[i] > a[m]$ er sann. Vi har to ytterpunkter. Hvis tabellens største verdi ligger først, vil $a[i] > a[m]$ aldri være sann og dermed blir x lik 0. Hvis alle verdiene er forskjellige og er sortert stigende, vil $a[i] > a[m]$ være sann hver gang. I det tilfellet blir x lik $n - 1$. Generelt vil x ligge et sted mellom disse ytterpunktene.

Eksempel 1.1.3: Tabellen i *Figur* 1.1.2 inneholder 15 tall. Sammenligningen $a[i] > a[m]$ vil være sann for denne tabellen hver gang vi finner en verdi som er større enn den største verdien vi tidligere har funnet. Det vil skje første gang når $i = 2$ og andre (og siste) gang når $i = 5$. Dermed får x verdien 2. Regnskapet over sier dermed at det blir utført $5 \cdot 15 + 2 = 77$ grunnleggende operasjoner for å finne den største verdien i tabellen.

Dominerende operasjoner Det er ikke vanlig å ta med alle operasjonene når en vurderer en algoritmes effektivitet. Det er ofte slik at én eller noen av operasjonene er viktigere, mer sentrale eller mer kostbare enn de andre. Slike operasjoner kalles *dominerende*. Eksempel: I *maks*-metoden i *Programkode* 1.1.2 er det å sammenligne verdier selve kjernen i algoritmen. De andre operasjonene er mer eller mindre en del av implementasjonen. Det å *sammenligne* er derfor den dominerende operasjonen.

```
if ( a[i] > a[m] ) ....
```

Figur 1.1.3 : Det å sammenligne er en dominerende operasjon i mange algoritmer

Men i andre algoritmer kan det være andre typer operasjoner som er dominerende. I numeriske algoritmer (dvs. algoritmer som inneholder tallberegninger) kan f.eks. det å *multiplisere* eller det å *dividere* være den dominerende operasjonen.

En algoritmes orden Resultatet av opptellingen (antall ganger den *dominerende* operasjonen utføres) avhenger av oppgavestørrelsen. I *maks*-metoden er den bestemt av antall verdier n (tabellengden). Den dominerende operasjonen ($a[i] > a[m]$) utføres $n - 1$ ganger. Det gir en lineær funksjon i n og vi sier at algoritmen er av *lineær orden* eller av *orden* n. Hvis funksjonen hadde vært kvadratisk i n, ville vi ha sagt at den var av *kvadratisk orden* eller av *orden* n^2. Lineær orden er generelt mer effektivt enn kvadratisk orden.

Opptellinger gir oss funksjoner med oppgavestørrelsen n som argument. De kan generelt ordnes i rekkefølge etter veksthastighet, dvs. hvor fort de vokser når argumentet n øker. Denne rekkefølgen kalles funksjonenes *asymptotiske* orden. *Tabell* 1.8.2 inneholder en del vanlige funksjoner (med n som argument). Jo lenger ned i tabellen, jo fortere vokser

funksjonen. Funksjoner i algoritmeanalyse vil normalt være en lineærkombinasjon av de fra *Tabell* 1.8.2 (og eventuelt andre). Anta at «opptellingen» for en bestemt algoritme gir oss funksjonen $f(n) = 3n^2 + 4n + \log_2(n) + 5$. Her er det leddet $3n^2$ som avgjør den asymptotiske oppførselen siden n^2 står lenger ned i *Tabell* 1.8.2 enn de andre. Leddet n^2 kalles funksjonens dominerende ledd. Vi ser bort fra de andre leddene (og fra eventuelle konstanter) og sier at denne algoritmen er av orden n^2. Dette blir på samme måte som for *maks*-metoden. Der fikk vi funksjonen $f(n) = n - 1$ og dermed orden n (og ikke $n - 1$).

I litteratur med en mer rigid fremstilling av algoritmeanalyse, brukes vanligvis notasjonen «stor O» (eng: big O). Med den vil *maks*-metoden være O(n) og den tenkte algoritmen i avsnittet rett over, $O(n^2)$. Men det at en algoritme er $O(g(n))$ der $g(n)$ f.eks. er en av funksjonene i *Tabell* 1.8.2, er egentlig en påstand om at $g(n)$ er en øvre grense. Dvs. at algoritmen i hvert fall ikke har en dårligere orden enn $g(n)$, men kan være bedre. Dermed er det formelt sett korrekt å si at en algoritme som er O(n) også er $O(n^2)$. I de tilfellene man har en eksakt formel for algoritmens «oppførsel» (med $g(n)$ som dominerende ledd) finnes det en mer presis notasjon, dvs. at den er $\Theta(g(n))$. Dette kalles «stor Θ»-notasjon (Θ = teta). Hos oss svarer i hovedsak «algoritmen er av orden $g(n)$» til at «algoritmen er $\Theta(g(n)$»

Det er ofte vanskelig å finne et eksakt funksjonsuttrykk for en algoritmes «oppførsel». Det gjelder spesielt i de tilfellene der «oppførselen» er forskjellig i det som kalles det beste, det gjennomsnittlige og det verste tilfellet. Da må vi ofte nøye oss med en tilnærming $f(n)$ (der $g(n)$ er dominerende ledd) som garantert ikke er «bedre» enn den korrekte oppførselen. Da kan vi si at algoritmen er $O(g(n))$ eller at den har orden $g(n)$ eller er bedre.

Den formelle definisjonen av «stor O»: En funksjon $f(n)$ er $O(g(n))$ hvis det finnes positive konstanter M og n_0 slik at $|f(n)| \leq M|g(n)|$ for alle $n \geq n_0$. I algoritmeanlyse vil slike funksjoner normalt representere en «opptelling» og dermed være positive, dvs. ha positive funksjonsverdier for $n > 0$. Dermed kan tallverditegnene tas vekk. Grafisk betyr dette at kurven (grafen) til $f(n)$ vil ligge på undersiden av den til $Mg(n)$ for alle $n \geq n_0$.

Ovenfor ble det påstatt at en funksjon som er O(n) også er $O(n^2)$. Hvorfor? La $f(n) = n$ og $g(n) = n^2$. Vi har $n \leq n^2$, $n \geq 1$. Med $n_0 = 1$ og $M = 1$ blir $f(n) \leq Mg(n)$ for alle $n \geq n_0$.

Disse matematiske begrepene om en algoritmes effektivitet kan være vanskelig å forstå ved første gangs lesning og krever også en viss matematisk innsikt. Du finner mer detaljert og grundig stoff om dette i *Delkapittel* 1.8 om *Algoritmeanalyse* eller på Wikipedia.

Algoritmeeffektivitet - oppsummering:

- En algoritme er en beskrivelse av de arbeidsoperasjonene som skal til for å løse en bestemt oppgave.
- Den har vanligvis en dominerende operasjon - f.eks. det å sammenligne.
- Vi kan «telle opp» antallet ganger den dominerende operasjonen utføres.
- Antallet kan ofte uttrykkes som en funksjon av oppgavens størrelse n. Denne funksjonen definerer algoritmens orden.
- Algoritmens orden er en målestokk for dens effektivitet.
- Jo «mindre» en algoritmes orden er, jo mer effektiv er den. F.eks. er lineær orden generelt mer effektivt enn kvadratisk orden. *Tabell* 1.8.2 viser de funksjonene som vi oftest kommer i kontakt med i algoritmeanalyse.
- *Maks*-metoden (*Programkode* 1.1.2) har lineær orden (orden n). Den er effektiv. Det er umulig å finne den største i en usortert tabell med færre enn n - 1 sammenligninger. Men koden kan optimaliseres noe (*Avsnitt* 1.1.4).

Oppgaver til Avsnitt 1.1.3

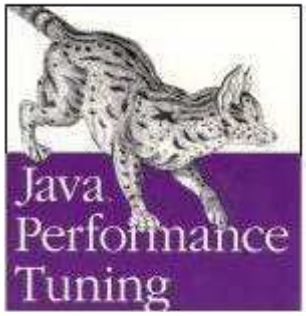
1. Et eksempel på en oppgave som kan løses på mange forskjellige måter er det å sortere verdiene i en tabell. Hvor mange sorteringsalgoritmer kjenner du til? På Wikipedia finner du en oversikt over sorteringsalgoritmer.
2. Gitt en tabell med verdiene 10, 5, 7, 2, 9, 1, 3, 8, 4, 6. Hvor mange grunnleggende operasjoner vil algoritmen i *Programkode* 1.1.2 utføre på denne tabellen. Se regnskapet over og *Eksempel* 1.1.3.
3. Som oppgave 2, men med verdiene 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.
4. Som oppgave 2, men med verdiene 1, 3, 2, 7, 5, 9, 6, 8, 10, 4.
5. Lag en metode `public static int[] minmaks(int[] a)`. Den skal ved hjelp av en *int*-tabell med lengde 2 returnere posisjonene til minste og største verdi i tabellen *a*. Hvis du har funnet at `m1` er posisjonen til den minste og `m2` til den største, kan du returnere tabellen *b* definert ved: `int[] b = {m1, m2};` Hvor mange sammenligninger bruker metoden din?
6. Utrykket $n!$ betyr n fakultet og er gitt ved $n! = n \cdot (n\text{-}1) \cdot \ \cdot \ \cdot 2 \cdot 1$. Lag en metode `long fak(int n)` som regner ut $n!$. Hvor mange multiplikasjoner utføres i metoden?

1.1.4 Optimalisering av programkode

Når en oppgave kan løses på flere måter, er det viktig å velge en god algoritme. Eller helst den beste hvis en slik finnes. En algoritme må implementeres for å kunne brukes. Første krav er at koden gjøres forståelig og lesbar. Men den bør også være optimalisert. Det er mange teknikker for å optimalisere kode og vi bør kjenne til noen av dem. Men det har lite for seg å lage «lur» kode hvis gevinsten er liten. Dessuten er moderne kompilatorer «smarte» ved at de i mange tilfeller optimaliserer koden vår mer enn vi kan klare selv.

Kan implementasjonen i *Programkode* 1.1.2 forbedres? Kan noen av operasjonene forenkles, fjernes eller erstattes av andre? Da er det spesielt viktig å se på operasjoner som inngår i løkker. I *Programkode* 1.1.2 er det tre løkkeoperasjoner som alle utføres $n - 1$ ganger. En av dem er sammenligningen `a[i] > a[m]` der tabelloperasjonene `a[i]` og `a[m]` inngår.

En tabelloperasjon er relativt sett en kostbar operasjon. For det første blir det sjekket om *a* er en *null*-tabell. Dernest blir indeksen sjekket for å se om den er innenfor lovlig område. Til slutt utføres en beregning for å finne minneadressen til det oppgitte tabellelementet. Med andre ord mange deloppgaver. Vår implementasjon burde derfor kunne bli bedre hvis antallet tabelloperasjoner reduseres. Et tilfelle som i hvert fall bør unngås er at samme tabellelement aksesseres mange ganger. En bedre løsning er da å kopiere tabellelementets verdi over i en hjelpevariabel og så aksessere hjelpevariablen. For som *Jack Shirazi* sier i boken *Java Performance Tuning*, «You can assume that array-element access is going to be slower then plain-variable access in almost every Java environment».



Jack Shirazi
Java Perfomance Tuning
O'Reilly, 2.ed. 2003

I *Programkode* 1.1.2 blir innholdet i hvert tabellelement fortløpende sammenlignet med den inntil da største verdien, dvs. verdien i posisjon *m*. Hvis vi finner en verdi som er større, blir *m* oppdatert. En mer effektiv teknikk er å bruke en hjelpevariabel *maksverdi* som hele tiden inneholder den største verdien. Da kan den brukes i sammenligningen. Koden blir slik (det som er nytt i forhold til *Programkode* 1.1.2 er satt i fet kursiv):

```
public static int maks(int[] a)   // versjon 2 av maks-metoden
{
  int m = 0;                 // indeks til største verdi
  int maksverdi = a[0];      // største verdi

  for (int i = 1; i < a.length; i++) if (a[i] > maksverdi)
  {
    maksverdi = a[i];        // største verdi oppdateres
    m = i;                   // indeks til største verdi oppdateres
  }
  return m;   // returnerer indeks/posisjonen til største verdi

} // maks
```

***Programkode* 1.1.4**

Nå har vi $a[i] > maksverdi$ istedenfor $a[i] > a[m]$, dvs. fra to tabelloperasjoner til én. Men hvis $a[i] > maksverdi$ er sann, blir tilordningen $maksverdi = a[i]$ utført og det betyr en ekstra tabelloperasjon. Har vi da fått færre tabelloperasjoner? Svaret på spørsmålet er avhengig av hvor mange ganger sammenligningen $a[i] > maksverdi$ er sann. Det viser seg heldigvis at i gjennomsnitt vil den være sann bare i få tilfeller. Vi ser nærmere på det i et senere avsnitt. Dermed burde vår nye versjon av *maks*-metoden være, i hvert fall teoretisk sett, noe mer effektiv enn den forrige. I tillegg må vi kunne si at koden vår fortsatt er forståelig og lett lesbar. I *Avsnitt* 1.1.10 skal vi gjøre testkjøringer for å avgjøre om forbedringen er signifikant (dvs. av betydning).

Eksempel 1.1.4: I *Eksempel* 1.1.3 fant vi at *maks*-metoden i *Programkode* 1.1.2 utførte 76 grunnleggende operasjoner på tabellen i *Figur* 1.1.2. Hva med *maks*-metoden i *Programkode* 1.1.4 ? Da blir det $2(n - 1)$ operasjoner i sammenligningen $a[i] > maksverdi$, men hver gang den er sann blir det nå utført 3 operasjoner (to tilordninger og en tabelloperasjon). Teller vi opp skulle det bli 67 grunnleggende operasjoner for tabellen i *Figur* 1.1.2.

Oppgaver til Avsnitt 1.1.4

1. Hvor mange grunnleggende operasjoner utfører *Programkode* 1.1.4 hvis tabellen *a* inneholder i) 10, 5, 7, 2, 9, 1, 3, 8, 4, 6 ii) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 og iii) 1, 3, 2, 7, 5, 9, 6, 8, 10, 4. Sammenlign med svarene i oppgave 2, 3 og 4 fra *Avsnitt* 1.1.3.

1.1.5 Optimalisering ved hjelp av en «vaktpost»

En militær vaktpost fra tidligere tider

Regnskapet i *Avsnitt* 1.1.3 viste at addisjonen `i++` ble utført $n - 1$ ganger og sammenligningen `i < a.length` ble utført n ganger. Addisjonen `i++` er det ikke mulig å fjerne eller forenkle.

Sammenligningen `i < a.length` er nødvendig for å hindre at indeksen går utenfor tabellen. En *ArrayIndexOutOfBoundsException* kastes hvis `i` blir lik `a.length`, og det må vi unngå. En «stoppverdi» lengst til høyre i tabellen kan fungere som et alternativ til det å utføre sammenligningen `i < a.length`. En slik verdi kalles på engelsk en *sentinel* og betyr ifølge ordboken: «*a soldier stationed as a guard to challenge all comers and prevent a surprise attack*». Vi vil noen steder bruke *stoppverdi* og andre steder *vaktpost* som navn på dette begrepet.

Hvis vi legger et stort heltall lengst til høyre i tabellen, vil for-løkken stoppe da `a[i]` ≥ *maksverdi* helt sikkert blir sann for siste tabellindeks `i`. Vi kan f.eks. bruke maksimumsverdien for datatypen `int`, dvs. tallet 2147483647 som vaktpost. Når en vaktpost legges bakerst, må vi ta vare på det som opprinnelig lå der. Flg. kode gjør dette:

```
public static int maks(int[] a)  // versjon 3 av maks-metoden
{
  int sist = a.length - 1;       // siste posisjon i tabellen
  int m = 0;                     // indeks til største verdi
  int maksverdi = a[0];          // største verdi
  int temp = a[sist];            // tar vare på siste verdi
  a[sist] = 0x7fffffff;          // legger tallet 2147483647 sist

  for (int i = 0; ; i++)         // i starter med 0
    if (a[i] >= maksverdi)       // denne blir sann til slutt
    {
      if (i == sist)             // sjekker om vi er ferdige
      {
        a[sist] = temp;          // legger siste verdi tilbake
        return temp >= maksverdi ? sist : m;   // er siste størst?
      }
      else
      {
        maksverdi = a[i];        // maksverdi oppdateres
        m = i;                   // m oppdateres
      }
    }
} // maks
```

***Programkode* 1.1.5**

I *Programkode* 1.1.5 blir tallet 2147483647 (maksimumsverdien for datatypen `int`) lagt sist i *a* som stoppverdi. Men denne verdien kan jo også ligge et annet sted som en ordinær verdi. Derfor må for-løkken ha `a[i] >= maksverdi` for å sikre at sammenligningen helt sikkert blir sann når vi kommer til siste posisjon. En konsekvens av å bruke >= istedenfor > er at hvis største verdi ligger flere steder, vil det være posisjonen til den siste av dem som returneres og ikke den første som tidligere. Se *Oppgave* 2.

Er *Programkode* 1.1.5 mer optimal enn de tidligere versjonene? Vi har fått inn `i == sist` som en ekstra sammenligning. Men den blir utført kun når `a[i] >= maksverdi` er sann og det vil skje gjennomsnittlig (se neste avsnitt) få ganger. Det betyr at denne siste versjonen utfører færre operasjoner enn forrige versjon siden sammenligningen `i < a.length` har forsvunnet. Den skulle derfor teoretisk sett være mer effektiv. Om den virkelig er mer effektiv, dvs. om den har et signifikant mindre tidsforbruk, blir det gitt svar på i *Avsnitt* 1.1.10. Der skal vi gjøre testkjøringer.

Teknikken med en vaktpost eller stoppverdi for å signalisere at vi har nådd slutten på noe, skal vi bruke flere ganger senere. Det er tilfeller der teknikken vil gi større gevinst enn her.

Oppgaver til Avsnitt 1.1.5

1. Sjekk at *Programkode* 1.1.5 gir korrekte resultater. Hva skjer hvis *a* har lengde 1 og hva hvis *a* er tom (lengde 0). Se også *Avsnitt* 1.1.7.
2. Gjør om *Programkode* 1.1.5 slik at posisjonen til den første av dem returneres hvis den største verdien ligger flere steder.
3. En vaktpostverdi må være større enn eller lik alle mulige tabellverdier. Men ikke alle typer har en maksverdi slik som `int` har. Vi kan isteden la tabellens siste posisjon hele tiden inneholde det samme som hjelpevariablen `maksverdi`. Dermed vil `a[i] >= maksverdi` helt sikker bli sann når `i` blir lik `sist`. Innfør denne idéen i *Programkode* 1.1.5.

1.1.6 Enkel algoritmeanalyse

Donald E. Knuth

Algoritmeanalyse (eng: analysis of algorithms) handler om å studere algoritmers egenskaper ved hjelp av matematiske og statistiske teknikker. *Donald E. Knuth* regnes som grunnleggeren av dette fagområdet. I 1967 var han ferdig med første bind av sitt berømte bokverk «*The Art of Computer Programming*». Der inngår *diskret matematikk* som det sentrale verktøyet.

I *Avsnitt* 1.1.4 var konklusjonen at vi måtte vite hvor mange ganger sammenligningen `a[i] > maksverdi` er sann i gjennomsnitt for å kunne avgjøre om *Programkode* 1.1.4 er mer optimal enn *Programkode* 1.1.2. Hvis vi antar at alle tallene i tabellen *a* er forskjellige, så er det faktisk mulig å finne en formel for det gjennomsnittlige antallet ganger testen `a[i] > maksverdi` er sann.

I *Programkode* 1.1.4 blir ett og ett tall (fra og med det andre tallet, dvs. indeks 1) hentet ut fra tabellen og sammenlignet med det tallet som inntil da var det største. Hvis det nye tallet er enda større blir testen sann. Det betyr at testen blir sann for hver gang vi finner et tall som er større enn det største av tallene foran.

Tabell 1: 4, **6**, 3, 5, **8**, 1, **9**, 2, **10**, 7
Tabell 2: 7, 5, 1, 6, **8**, 4, 3, **10**, 2, 9
Tabell 3: 10, 3, 8, 2, 6, 9, 1, 5, 4, 7
Tabell 4: 1, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **10**

Her har vi fire tabeller som alle inneholder tallene fra 1 til 10, men i forskjellige rekkefølger. Det første tallet har jo ingen tall foran seg. Det første mulige tallet som er større enn det største av tallene foran, er derfor det andre tallet. I *Tabell* 1 ser vi at fire av tallene har den egenskapen at det er større enn det største av tallene foran. Det er tallene 6, 8 , 9 og 10. I *Tabell* 2 er det bare 8 og 10 som har den egenskapen. I *Tabell* 3 er det ingen slike tall. Det kommer av at 10 ligger først og ingen av de andre er større enn 10. I *Tabell* 4 har alle tallene, fra og med det andre, denne egenskapen.

Vi er interessert i det gjennomsnittlige antallet ganger et tall er større enn det største foran. Det kunne vi finne for tilfellet 10 forskjellige tall. Vi måtte da først sette opp alle mulige rekkefølger og så, for hver slik rekkefølge, finne antallet tall som er større enn det største av tallene foran. Gjennomsnittet finner vi ved å summere alle antallene og dele summen på antallet forskjellige rekkefølger. En rekkefølge av tallene fra 1 til 10 kalles en *permutasjon* og det finnes 10! = 3.628.800 forskjellige av dem. Vi forstår at selv med så få som 10 tall er det umulig å gjøre dette for hånd. Men det er enkelt å lage et program som gjør det.

Vi skal imidlertid klare å regne ut dette for hånd i noen enkle tilfeller, dvs. for 2, 3 eller 4 forskjellige verdier. Men før vi gjør det kan du jo teste din egen intuisjon. Hvis vi har en tabell med f.eks. 100.000 forskjellige og tilfeldige tall, hvor mange av dem tror du vil være større enn det største av tallene foran? Skriv ned forslaget ditt!

Gitt tallene 1 og 2. Da får vi permutasjonene 1,**2** og 2,1. I første tilfelle er det ett tall som er større enn det største foran, og i andre ingen. Gjennomsnittet blir (1 + 0)/2 = 1/2.

1 **2 3 4**	3
1 **2 4** 3	2
1 **3** 2 **4**	2
1 **3 4** 2	2
1 **4** 2 3	1
1 **4** 3 2	1
2 1 **3 4**	2
2 1 **4** 3	1
2 **3** 1 **4**	2
2 **3 4** 1	2
2 **4** 1 3	1
2 **4** 3 1	1
3 1 2 **4**	1
3 1 **4** 2	1
3 2 1 **4**	1
3 2 **4** 1	1
3 **4** 1 2	1
3 **4** 2 1	1
4 1 2 3	0
4 1 3 2	0
4 2 1 3	0
4 2 3 1	0
4 3 1 2	0
4 3 2 1	0

Hvis vi har tallene 1, 2 og 3 får vi de 6 permutasjonene

1,**2**,**3** 1,**3**,2 2,1,**3** 2,**3**,1 3,1,2 3,2,1

Vi får henholdsvis 2, 1, 1, 1, 0 og 0 for antallet ganger det er et tall som er større enn det største foran. Gjennomsnittet blir (2 + 1 + 1 + 1 + 0 + 0)/6 = 5/6. Her observerer vi at 5/6 = 1/2 + 1/3.

I første kolonne i tabellen til venstre er alle de 24 forskjellige permutasjonene av tallene 1, 2, 3 og 4 satt opp. For hver permutasjon teller vi opp antallet tall som er større enn det største foran. Resultatet er satt opp i den andre kolonnen. Gjennomsnittet finner vi ved å summere og dele resultatet med 24. Svaret blir: 26/24 = 13/12 = 1/2 + 1/3 + 1/4. Dermed kan flg. påstand være rimelig:

Setning 1.1.6 a) *I en tabell med n* ($n > 1$) *forskjellige tall er i gjennomsnitt* $1/2 + 1/3 + 1/4 + \ldots + 1/n$ *av dem større enn det største av de foran.*

Det er åpenbart det samme om vi bruker *n* forskjellige tall eller tallene fra 1 til *n*. Dermed har vi vist (se ovenfor) at *Setning* 1.1.6 *a)* er sann for $n = 2$, 3 og 4. Det å vise at den er sann for alle *n* krever et induksjonsbevis. Det finner du, hvis du er interessert, i *Avsnitt* 1.1.13.

Vi trenger en tilnærmingsverdi for summen i *Setning* 1.1.6 *a)*. Det *n*-te *harmoniske tall* er definert som summen av de inverse heltallene fra 1 til *n* og betegnes med H_n :

Leonhard Euler
1707 - 1783

$$H_n = 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n$$

La $\log(n)$ være den naturlige logaritmen til *n* (dvs. logaritmen med *e* som grunntall). Det er kjent at forskjellen mellom H_n og $\log(n)$ nærmer seg en grense når *n* vokser. Grensen kalles *Eulers konstant* og er med 3 desimalers nøyaktighet lik 0,577. For store *n* kan vi derfor sette:

$$H_n \approx \log(n) + 0{,}577$$

Summen i *Setning* 1.1.6 *a)* mangler 1-tallet. Den blir dermed lik $H_n - 1$.

Setning 1.1.6 b) *I en tabell med n forskjellige tall (n stor) er i gjennomsnitt* $\log(n) - 0{,}423$ *av dem større enn det største av tallene foran.*

Hvis n er lik 100.000 blir log(n) – 0,423 tilnærmet lik 11,1. Det betyr at hvis det er 100.000 forskjellige tall, vil det gjennomsnittlig bare være 11,1 av dem som er større enn det største av de foran. Mange vil mene at det var overraskende få. Du ble ovenfor bedt om å foreslå et antall. Hva var ditt forslag? Tabellen under viser resultatet for noen forskjellige verdier av n:

n	10	100	1.000	10.000	100.000	1.000.000	10.000.000
gj.ant.	1,9	4,2	6,5	8,8	11,1	13,4	15,7

Tabell 1.1.6

Hensikten med analysen var å finne antallet ganger sammenligningen `a[i] > maksverdi` er sann i *Programkode* 1.1.4. Dette er det samme som antallet ganger en tabell inneholder et tall som er større enn det største av tallene foran. Analysen viser at dette antallet i gjennomsnitt er svært lite hvis alle verdiene er forskjellige. Hvis tabellen har like verdier blir antallet enda mindre. Det merarbeidet som skjer i *Programkode* 1.1.4 (og i *Programkode* 1.1.5) hver gang `a[i] > maksverdi` er sann, består derfor bare av noen få operasjoner. Konklusjonen er derfor at det utføres færre grunnleggende operasjoner i *Programkode* 1.1.4 (og i *Programkode* 1.1.5) enn i *Programkode* 1.1.2, og den er derfor, i hvert fall teoretisk sett, mer effektiv.

Når vi skal vurdere effektiviteten til en algoritme skiller vi normalt mellom gjennomsnittlig effektivitet og effektiviteten i det mest ugunstige tilfellet. I gjennomsnitt er *Programkode* 1.1.4 teoretisk sett bedre enn *Programkode* 1.1.2. Det mest ugunstige tilfellet får vi når tallene er forskjellige og sortert stigende. Da vil `a[i] > maksverdi` være sann hver gang, og setningen `maksverdi = a[i]` blir utført hver gang. I dette tilfellet vil *Programkode* 1.1.4 ikke være bedre enn *Programkode* 1.1.2. Faktisk litt dårligere.

Oppgaver til Avsnitt 1.1.6

1. Vis at *Setning* 1.1.6 *a)* stemmer for n = 5. Se på alle de 120 forskjellige permutasjonene av tallene 1, 2, 3, 4 og 5. Flg. observasjon kan forenkle arbeidet: De 24 permutasjonene der 5 står først, inneholder ingen tall som er større enn alle foran. Blant de 24 der 5 står som nr 2, vil det være nøyaktig ett tall (5 selv) som er større enn de foran. Osv.
2. La a(n,k) være antallet permutasjoner av 1, . . , n som har nøyaktig k tall som er større enn det største foran. F.eks. er a(3,1) = 3 fordi de tre permutasjonene (1,3,2), (2,1,3) og (2,3,1) (og ingen andre) har nøyaktig ett tall som er større enn alle foran. Finn a(3,0), a (3,2), a(4,0), a(4,1), a(4,2) og a(4,3).
3. Lag metoden `public static double harmonisk(int n)`. Metoden skal ved hjelp av en løkke regne ut (og returnere) det n-te harmoniske tallet.
4. Lag metoden `public static double euler(int n)`. Den skal returnere differansen mellom H_n og log(n). I Java gir Math.log(n) oss den naturlige logaritmen til n. Hvor stor må n være for at `euler(n)` skal returnere et tall som starter med 0,577 som de tre første desimalene?
5. Lag en programbit som gir resultatene i *Tabell* 1.1.6.

1.1.7 Testing for korrekthet – enhetstesting

Det første kravet vi stiller til en algoritme er at den er korrekt. En algoritme som ikke virker eller som gir oss et galt resultat, har ingen verdi. En slik algoritme sies å ha feil eller «bugs».

«Bugs»

En viktig teknikk i programvaretesting har fått navnet *enhetstesting* (eng: unit testing). En *enhet* er da den minste testbare delen av et program. Det kan være en klasse eller like gjerne en enkelt metode siden en klasse normalt har mange metoder. På engelsk heter det: «The primary goal of unit testing is to take the smallest piece of testable software in the application, isolate it from the remainder of the code, and determine whether it behaves exactly as you expect.»

En kan kode metoden først og teste etterpå. Alternativt kan en lage testen først f.eks. som en programbit (eng: stub) og så lage koden etterpå. Hvis en gjør dette systematisk, kalles det *testdrevet utvikling* (eng: test-driven development).

I dette emnet (Algoritmer og datstrukturer) handler det i stor grad om å lage algoritmer/metoder for ulike formål. Dermed blir normalt en metode den minste testbare enheten. En metode skal utføre en konkret oppgave. Derfor er det kanskje enklest for oss å kode først og teste etterpå.

En enkel teknikk kalles *papirtesting*. Da skriver vi opp en serie testverdier eller forutsetninger og så sjekker vi nøye for hvert tilfelle, ved hjelp av programkoden, at alle operasjoner og programsetninger gjør det de skal. Dette kan utvides til at vi lager en programbit der det testes ved hjelp av bestemte verdier/forutsetninger og ved hjelp av tilfeldig valgte verdier/forutsetninger.

Vi må passe på at alle mulige tilfeller tas med. Det vil si:

1. de normale eller vanlige tilfellene
2. de spesielle, men likevel lovlige tilfellene
3. de ulovlige tilfellene

Vi har laget tre versjoner av *maks*-metoden. De er kodet litt forskjellig, men skal jo gi nøyaktig samme resultat for alle tilfeller. Derfor må en test lages uavhengig av hvordan metoden er kodet. Det er kun «oppførselen» som skal testes. Det er av og til et problem med de «ulovlige» tilfellene. Hva burde en *maks*-metode gjøre hvis den f.eks. kalles på en tom tabell? Vi har laget det slik at det da kastes et unntak. Men det hadde også vært mulig å signalisere gjennom en returverdi at vi har et ulovlig tilfelle. Enkelte ganger gjør man begge deler. Da lages det to metoder. De oppfører seg identisk for «lovlige» tilfeller, men ulikt for de «ulovlige». F.eks. ved å kaste et unntak i den ene versjonen og ved å bruke en spesiell returverdi i den andre. Flere klasser i Java har dobbelt sett av metoder av denne typen.

Vi har egentlig ikke satt opp nøyaktige krav til en *maks*-metodes «oppførsel». Men det må vi gjøre før vi begynner testingen. Vi krever at metoden `public static int maks(int[] a)` skal returnere posisjonen til den største verdien i en ikke-tom heltalstabell *a*. Hvis den største verdien forekommer flere steder, skal posisjonen til den første av dem returneres. Hvis tabellen *a* er tom, skal det kastes en `NoSuchElementException` og hvis *a* er null, skal det kastes en `NullPointerException`.

Hva er vanlige tilfeller? Det må være at tabellen *a* inneholder en del verdier og er generelt usortert. Den største kan ligge først, til slutt eller et sted mellom. Videre kan den største verdien forekomme flere steder. Spesialtilfellene kan f.eks. være at tabellen har kun én verdi, har kun to verdier som er ulike eller kun to verdier som er like. De ulovlige tilfellene er at *a* er tom eller null.

Hvis en skal lage en testmetode som sjekker at metodens «oppførsel» er korrekt i både vanlige og uvanlige tilfeller og i ulovlige tilfeller, blir det det normalt lang kode. Ofte vil det være mer arbeid å lage testmetoden enn metoden som skal testes. Men fordelen er at hvis en senere gjør endringer (eller nye versjoner) i metoden, kan testmetoden brukes om igjen.

Vi må fjerne eventuelle «bugs» i koden

Vi nøyer oss her med å teste *maks*-metoden for et vanlig tilfelle og et ulovlig tilfelle. I oppgavene vil du bli bedt om å utvide testmetoden.

```
public static void makstest()
{
  int[] a = {8,3,5,7,9,6,10,2,1,4};  // maksverdien 10 er i posisjon 6

  if (maks(a) != 6)  // kaller maks-metoden
    System.out.println("Kodefeil: Gir feil posisjon for maksverdien!");

  a = new int[0];  // en tom tabell, lengde lik 0
  boolean unntak = false;

  try
  {
    maks(a);  // kaller maks-metoden
  }
  catch (Exception e)
  {
    unntak = true;
    if (!(e instanceof java.util.NoSuchElementException))
      System.out.println("Kodefeil: Feil unntak for en tom tabell!");
  }

  if (!unntak)
    System.out.println("Kodefeil: Mangler unntak for en tom tabell!");
}
```

Programkode* 1.1.7 *a)

Returverdi som feilsignal *Programkode* 1.1.7 *a)* tester om *maks*-metoden kaster et unntak for en tom tabell og om unntaket er av rett type. Dette vil stemme for den versjonen av maks-metoden som er gitt i *Programkode* 1.1.2, men ikke for de to versjonene i *Programkode* 1.1.4 og *Programkode* 1.1.5. Se *Oppgave* 1 - 3. Det er imidlertid mange situasjoner der det ikke er gunstig å kaste et unntak for en tom tabell. En mulighet er da å signalisere dette ved hjelp av en bestemt returverdi. I maks-metoden kunne vi f.eks. vi bruke -1 (eller en annen negativ verdi) siden -1 ikke kan være en tabellindeks. Men hvis vi har en maks-metode som gir oss den største verdien (dvs. verdien og ikke posisjonen), kan ikke -1 brukes. Hvis tabellen *a* inneholder negative verdier, vil vi da ikke kunne vite om -1 er den største av dem eller om -1 signaliserer en «feil».

Vi kan løse dette problemet ved å la metoden returnere et objekt. Det skal inneholde korrekt resultat hvis alt fungerer som normalt, men ingenting hvis det oppstår en feil. Vi kan lage en egen klasse for dette, men det er unødvendig siden det allerede finnes en slik klasse i Java.

Det er `OptionalInt`. Det engelske ordet *optional* kan oversettes med valgfri eller frivillig. Her er imidlertid `OptionalInt` «a container object which may or may not contain an int value». Klassen har ingen offentlig konstruktør, men derimot to konstruksjonsmetoder, dvs. to statiske metoder som begge returnerer en instans av klassen. Flg. metode finner indeks til største verdi i en tabell og bruker en `OptionalInt` som returverdi (se også *Avsnitt* 1.1.11):

```
// krever import java.util.OptionalInt;

public static OptionalInt maks(int[] a)           // indeks til største verdi
{
  if (a.length < 1) return OptionalInt.empty();   // en konstruksjonsmetode

  int m = 0, maksverdi = a[0];                    // startindeks og verdi

  for (int i = 1; i < a.length; i++)              // starter med i = 1
  {
    if (a[i] > maksverdi)
    {
      m = i; maksverdi = a[i];                    // oppdaterer
    }
  }

  return OptionalInt.of(m);                       // en konstruksjonsmetode
}
```

Programkode* 1.1.7 *b)

Flg. eksempel viser hvordan metoden `maksverdi` kan brukes:

```
int[] a = {8,3,5,7,9,6,10,2,1,4}, b = {};  // to tabeller

System.out.println(maks(a));       // utskrift: OptionalInt[6]
System.out.println(maks(b));       // utskrift: OptionalInt.empty
```

Programkode* 1.1.7 *c)

Metoden `maksverdi` returnerer en instans av `OptionalInt`. Det er dens `toString`-metode som brukes over. Det er også mulig å sjekke om den har innhold (metoden `isPresent`) og i såfall hente dens innhold (metoden `getAsInt`). Se *Oppgave* 5.

Oppgaver til Avsnitt 1.1.7

1. Lag et main-program med metoden `makstest` der *Programkode* 1.1.2 testes. Hva skjer?
2. Utvid metoden `makstest`. Bruk tabeller der den største er først, er sist og forekommer flere steder. Bruk en tabell med kun én verdi, kun to verdier som er ulike og kun to verdier som er like. Lag en test for en null-tabell. La også `makstest` returnere antall feil.
3. Bruk `makstest` på de to versjonene i *Programkode* 1.1.4 og 1.1.5. Da vil det bl.a. komme melding om at det kastes feil unntak for en tom tabell. Hvilket unntak kastes? Gjør om koden slik at rett unntak kastes. *Programkode* 1.1.5 gir feil svar hvis den største verdien ligger flere steder. Rett opp koden! Se også *Oppgave* 2 i *Avsnitt* 1.1.5.
4. Sett deg inn i begrepet `assert` i Java.
5. Lag et program som kjører *Programkode* 1.1.7 *c)*. Gjør om koden slik at resultatet fanges, dvs. bruk koden: `OptionalInt m = maks(a);` Sjekk deretter om resultat har innhold (`isPresent`) og hvis den har det, skriv ut verdien ved å bruke `getAsInt`. Finn ut hvilke andre metoder klassen `OptionalInt` har.

1.1.8 Generering av testverdier

Hvis vi skal teste effektiviteten til algoritmer som arbeider med tabeller, trenger vi ofte store tabeller med et tilfeldig innhold. I noen tilfeller kreves det at alle verdiene er forskjellige og i andre tilfeller er det aktuelt med like verdier. Hvis alle verdiene skal være forskjellige, kan vi like gjerne la det være tallene fra 1 til *n*. Java har flere metoder som genererer tilfeldige enkelttall. Et eksempel er metoden `int nextInt(int n)` i *Random* som returnerer et tilfeldig heltall fra mengden {0, 1, 2, . . . , *n* - 1}.

Det å lage en tilfeldig heltallstabell med tallene fra 1 til *n* (en permutasjon) trengs så ofte at det lønner seg å lage en metode for det. Det gir oss også muligheten til å demonstrere at en enkel idé ikke nødvendigvis fører til en effektiv metode. Ofte må en tenke annerledes for å få til det. Flg. kode er første forsøk:

```
import java.util.*;

public static int[] randPerm(int n)  // en versjon som ikke virker
{
  Random r = new Random();       // en randomgenerator
  int[] a = new int[n];          // en tabell med plass til n tall

  for (int i = 0; i < n; i++)
    a[i] = r.nextInt(n) + 1;     // tabellen fylles med tall

  return a;                      // tabellen returneres
}
```

Programkode* 1.1.8 *a)

Metoden gir imidlertid ikke det vi ønsker. Flg. kodebit gir et eksempel på hva som vil skje:

```
System.out.println(Arrays.toString(randPerm(10)));

// Utskrift: [10, 9, 8, 4, 1, 7, 5, 9, 4, 6]
```

Når metoden kalles med *n* = 10, får vi kallene `r.nextInt(10) + 1`. Det gir heltall fra 1 til 10, men ikke, slik som utskriften også viser, nødvendigvis forskjellige tall. Metoden `nextInt` «trekker ut» tall med såkalt tilbakelegging, dvs. at alle «tallene som det velges fra» er tilgjengelige ved neste kall på `nextInt`.

En enkel måte å forbedre dette på kunne være å sjekke det tallet som `nextInt` gir oss. Hvis det er «trukket ut» før, «kaster» vi det og «trekker» et nytt tall. For å avgjøre om et tall allerede er «trukket ut», kan vi f.eks. lete blant tallene som vi har tatt vare på. Tabellen *a* i *Programkode* 1.1.8 *a)* fungerer som et «lager». Anta at vi har klart å få fem forskjellige tall fra 1 til 10. Da vil tabellen *a* f.eks. kunne se slik ut:

7	2	5	1	9					
0	1	2	3	4	5	6	7	8	9

Ved hvert nytt kall på `nextInt` leter vi gjennom de tallene som er «godtatt». Hvis tabellen *a* inneholder *i* tall som er «godtatt», leter vi gjennom dem (intervallet *a*[0 : *i* >) ved hjelp av en for-løkke. Finner vi det nye tallet der, hopper vi ut av for-løkken:

```
public static int[] randPerm(int n)  // virker, men er svært ineffektiv
{
  Random r = new Random();       // en randomgenerator
  int[] a = new int[n];          // en tabell med plass til n tall

  for (int i = 0; i < n; )       // vi skal legge inn n tall
  {
    int k = r.nextInt(n) + 1;    // trekker et nytt tall k

    int j = 0;
    for ( ; j < i; j++)          // leter i intervallet a[0:i>
    {
      if (a[j] == k) break;      // stopper hvis vi har k fra før
    }
    if (i == j) a[i++] = k;      // legger inn k og øker i
  }
  return a;                      // tabellen returneres
}
```

Programkode* 1.1.8 *b)

Bruker vi denne versjonen av `randPerm`, vil flg. kodebit vise at vi nå får en tabell med bare forskjellige tall, dvs. en tilfeldig permutasjon av tallene fra 1 til 10:

```
System.out.println(Arrays.toString(randPerm(10)));

// Utskrift: [7, 2, 5, 1, 9, 6, 10, 8, 4, 3]
```

Programkode 1.1.8 *b)* fungerer, men har alvorlige svakheter. Etter som *a* fylles opp, tar det lenger og lenger tid å avgjøre om et tall finnes fra før. I tillegg synker, for hvert nytt tall, sannsynligheten for å få et av de manglende tallene. Mangler f.eks. kun ett tall, er den lik $1/n$. Da må `nextInt` i gjennomsnitt kalles n ganger for å få nettopp det tallet, i verste fall aldri. I gjennomsnitt er algoritmen av orden $n^2 \log(n)$. Se *Oppgave* 1 - 3.

En av svakhetene kan lett elimineres. Istedenfor å lete i tabellen for å avgjøre om vi allerede har tallet, kan vi gjøre et oppslag i en boolsk tabell. Da trekker vi indekser k, dvs. tall fra 0 til $n - 1$. Hvis tabellen inneholder *false* på indeks k, betyr det at tallet $k + 1$ er nytt. I den boolske tabellen settes det *true* på indeks k og $k + 1$ legges inn i tabellen a:

```
public static int[] randPerm(int n)  // virker, men er ineffektiv
{
  Random r = new Random();         // en randomgenerator
  int[] a = new int[n];            // en tabell med plass til n tall
  boolean[] har = new boolean[n];  // en boolsk tabell

  for (int i = 0; i < n; )         // vi skal legge inn n tall
  {
    int k = r.nextInt(n);          // trekker en indeks k
    if (har[k] == false)           // sjekker
    {
      har[k] = true;               // oppdaterer den boolske tabellen
      a[i++] = k + 1;              // legger inn k + 1 i a
    }
  }
  return a;                        // tabellen returneres
}
```

Programkode* 1.1.8 *c)

Programkode 1.1.8 *c)* er vesentlig bedre enn *Programkode* 1.1.8 *b)*. I *Oppgave* 4 - 7 blir du bl.a. bedt om å vise at den har orden $n\log(n)$ i gjennomsnitt. Men i verste fall vil den aldri gjøre seg ferdig. Det er imidlertid mulig å lage en metode som gir oss en tilfeldig permutasjon av tallene fra 1 til n og som alltid er av orden n, men da må vi tenke på en annen måte.

La $n = 10$. Vi tenker oss at tallene fra 1 til 10 ligger i en «beholder». Så trekkes et av tallene og legges til side. Poenget er at tallet vi trakk ikke legges tilbake i «beholderen». Med andre ord skal vi gjøre det som heter å trekke uten tilbakelegging. Neste gang vi trekker er det kun 9 tall i «beholderen». Tallet legger vi «ved siden av» det vi trakk først. Så trekker vi et nytt tall blant de 8 som er igjen. Osv. inntil alle tallene er trukket ut. Kan denne idéen kodes på en enkel måte?

Vi kan bruke en tabell som «beholder» og fylle den med tallene fra 1 til 10. Rekkefølgen er likegyldig og da gjør vi det som er enklest, dvs. å legge dem inn i stigende rekkefølge:

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9

Vi tar ut tall fra tabellen ved å trekke tilfeldige indekser. Kallet `nextInt(10)` gir oss en indeks fra 0 til 9. La oss si vi fikk indeksen 6. På den indeksen ligger tallet 7 som vi «fjerner»:

1	2	3	4	5	6		8	9	10
0	1	2	3	4	5	6	7	8	9

Hvor skal vi nå gjøre av tallet 7? Vi kan legge det bakerst. Men der ligger det jo et tall. Men det kan vi flytte til den ledige plassen vi har fått. Dermed:

1	2	3	4	5	6	10	8	9	7
0	1	2	3	4	5	6	7	8	9

I fortsettelsen skal vi tenke oss at «beholderen» som vi trekker fra består av den «hvite» delen av tabellen og at den «grå» delen inneholder de tallene som er fjernet. Vi trekker så en indeks fra 0 til 8 (kallet `nextInt(9)`). La oss si det ble 2. På indeks 2 ligger tallet 3. Når 3 fjernes, blir plassen ledig. Dit flytter vi det bakerste tallet i «beholderen» (det siste tallet i den «hvite» delen). Dermed kan 3 erstatte det tallet og få «grå» bakgrunn. Dvs. slik:

1	2	9	4	5	6	10	8	3	7
0	1	2	3	4	5	6	7	8	9

Vi fortsetter på denne måten. Kallet `nextInt(8)` gir en tilfeldig indeks fra 0 til 7. Hvis det f.eks. ble 2 igjen, får tabellen flg. innhold:

1	2	8	4	5	6	10	9	3	7
0	1	2	3	4	5	6	7	8	9

Generelt får vi at hvis $a[0:k]$ utgjør «beholderen» (den «hvite» delen av a), fjerner vi et tall fra «beholderen» ved å fjerne tallet på den indeksen i som kallet `nextInt(k + 1)` gir oss. Tallet tar vi vare på ved at det bytter plass med tallet på indeks k. I noen tilfeller betyr det at tallet bytter plass med seg selv. Vi kan lage en metode for denne plassbyttingen:

```
public static void bytt(int[] a, int i, int j)
{
  int temp = a[i]; a[i] = a[j]; a[j] = temp;
}
```

Programkode 1.1.8 d)

Dermed kan vi lage en ny og effektiv versjon (orden *n*) av metoden `randPerm`:

```
public static int[] randPerm(int n)  // en effektiv versjon
{
  Random r = new Random();          // en randomgenerator
  int[] a = new int[n];             // en tabell med plass til n tall

  Arrays.setAll(a, i -> i + 1);     // legger inn tallene 1, 2, . , n

  for (int k = n - 1; k > 0; k--)   // løkke som går n - 1 ganger
  {
    int i = r.nextInt(k+1);         // en tilfeldig tall fra 0 til k
    bytt(a,k,i);                    // bytter om
  }

  return a;                         // permutasjonen returneres
}
```

Programkode 1.1.8 e)

Gir *Programkode* 1.1.8 *e)* en tilfeldig permutasjon? Ja! Hver gang et tall «trekkes ut» har alle tallene det kan velges mellom (så sant `nextInt` gir tilfeldige indekser), samme sannsynlighet for å bli trukket ut. Metoden har orden *n* fordi den inneholder en løkke som går nøyaktig *n* – 1 ganger og i hver iterasjon er det kun et kall på `nextInt` og en ombytting av tabellverdier. Så nå har vi funnet den «ultimate» versjonen av `randPerm`-metoden.

Metoden i *Programkode* 1.1.8 *e)* returnerer en ny tabell hver gang den kalles. Noen ganger ganger er det mer aktuelt å permutere eller omstokke (eng: shuffle) verdiene i en tabell vi allerede har. Flg. metode stokker om på verdiene i tabellen *a*:

```
public static void randPerm(int[] a)  // stokker om a
{
  Random r = new Random();     // en randomgenerator

  for (int k = a.length - 1; k > 0; k--)
  {
    int i = r.nextInt(k + 1);  // tilfeldig tall fra [0,k]
    bytt(a,k,i);
  }
}
```

Programkode 1.1.8 f)

Programkode 1.1.8 *f)* kan f.eks. brukes slik:

```
int[] a = {6,8,1,4,2,10,7,9,3,5};
Tabell.randPerm(a);
System.out.println(Arrays.toString(a));
```

Programkode 1.1.8 g)

Oppgaver til Avsnitt 1.1.8

1. Kjør den kodebiten som kommer rett etter *Programkode* 1.1.8 *b)* flere ganger og sjekk at det hele tiden kommer ut permutasjoner av tallene fra 1 til 10.
2. Flg. kodebit viser hvor lang tid det tar å generere en tilfeldig permutasjon av tallene fra 1 til *n* ved å bruke *Programkode* 1.1.8 *b)*:

```
int n = 10000;
long tid = System.currentTimeMillis();
int[] a = randPerm(n);
tid = System.currentTimeMillis() - tid;
System.out.println(tid);
```

   Hvor mange millisekunder bruker denne kodebiten på din maskin? Øk verdien på *n* slik at kodebiten bruker ca. 5 sekunder (5000 millisekunder).
3. Vis at metoden i *Programkode* 1.1.8 *b)* er av orden $n^2 \log(n)$ i gjennomsnitt.
4. Sjekk at `randPerm`-versjonen i *Programkode* 1.1.8 *c)* virker som den skal. Se *Oppgave* 1. Du kan f.eks. omdøpe den første til `randPerm1` og den andre til `randPerm2`.
5. Gjør som i *Oppgave* 2, men bruk `randPerm`-versjonen i *Programkode* 1.1.8 *c)*. Er det stor forskjell på de to versjonene?
6. Vis at `randPerm`-versjonen i *Programkode* 1.1.8 *c)* er av orden $n \log(n)$ i gjennomsnitt.
7. I `randPerm`-versjonen i *Programkode* 1.1.8 *c)* brukes det en boolsk hjelpetabell. Gjør om metoden slik at den bruker samme idé, men uten hjelpetabellen. Første kall på `nextInt(n)` gir en indeks *k*. Vi legger så inn 1 i *a*[*k*]. Neste kall på `nextInt(n)` gir en ny indeks *k*. Hvis *a*[*k*] ikke er 0, kaller vi `nextInt(n)` på nytt. Hvis derimot *a*[*k*] er 0, legger vi inn 2 i *a*[*k*]. Osv. til hele *a* er fylt opp.
8. Sjekk at den versjonen av `randPerm` som står i *Programkode* 1.1.8 *e)* virker som den skal. Se *Oppgave* 1. Du kan f.eks. omdøpe den første versjonen til `randPerm1`, den andre til `randPerm2` og denne til `randPerm3`.
9. Gjør som i *Oppgave* 2, men bruk `randPerm`-versjonen i *Programkode* 1.1.8 *e)*. Er det stor forskjell på de tre versjonene? Hvilken av dem bør vi ta vare på og bruke senere når vi trenger en slik metode?
10. I *Programkode* 1.1.8 *e)* blir tallene som «fjernes» fra «beholderen» (tabellens «hvite» del) isteden lagt over i den «grå» delen. Lag en versjon av metoden der den «grå» delen er den venstre delen av tabellen og den «hvite» delen den høyre delen.
11. Lag `void randPerm(int[] a, int v, int h)` slik at den stokker om intervallet *a*[*v*:*h*] i tabellen *a*. Resten av tabellen skal være uberørt. Se *Programkode* 1.1.8 *f)*.
12. Tar vi ut fortløpende (uten tilbakelegging) *k* tilfeldige tall fra 1 til *n* får vi et *ordnet k-utvalg* (eller en *k-permutasjon*). Lag metoden `int[] randPerm(int n, int k)`. Den skal returnere et tilfeldig *ordnet k-utvalg* (0 <= *k* <= *n*).
13. Java har ingen ferdig metode som lager en tilfeldig heltallstabell. En omvei: Legg tallene fra 1 til *n* i en `ArrayList<Integer>`, bruk metoden `shuffle` i klassen `Collections` og hent så tallene fra listen og legg dem over i en int-tabell. Prøv dette!

1.1.9 Det harmoniske tallet H_n

I *Avsnitt* 1.1.6 ble det satt frem som en påstand at i en tabell med n forskjellige verdier, ville i gjennomsnitt $H_n - 1 \approx \log(n) - 0{,}423$ av dem være større enn det største av tallene foran. Vi kan sjekke denne påstanden ved å generere tilfeldige permutasjoner av tallene fra 1 til n, og så, ved å «telle opp», finne antallet tall av den typen i hver permutasjon.

```
public static int antallMaks(int[] a)    // teller opp i a
{
  int antall = 0;            // antall tall
  int maksverdi = a[0];

  for (int i = 1; i < a.length; i++)    // går gjennom tabellen a
  {
    if (a[i] > maksverdi)    // a[i] er større enn største foran
    {
      antall++;              // har funnet et nytt tall
      maksverdi = a[i];      // oppdaterer maksverdi
    }
  }

  return antall;    // de som er større enn det største foran
}
```

Programkode* 1.1.9 *a)

For å få et kjørbart program setter vi metodene *bytt*, *randPerm* og *antallMaks* inn i en klasse. Den kan f.eks. hete `class Program`. I tillegg trengs en *main*-metode:

```
import java.util.*;

public class Program
{
  // 1. Metoden bytt fra Programkode 1.1.8 d) skal inn her
  // 2. Metoden randPerm fra Programkode 1.1.8 e) skal inn her
  // 3. Metoden antallMaks fra Programkode 1.1.9 a) skal inn her

  public static void main(String ... args)   // hovedprogram
  {
    int n = 100_000;  // tabellstørrelse
    System.out.println(antallMaks(randPerm(n)));
  }
}
```

Programkode* 1.1.9 *b)

Da dette programmet ved ett tilfelle ble kjørt 10 ganger kom disse tallene ut: 18, 16, 13, 7, 7, 13, 15, 8, 5 og 12. Gjennomsnittsverdien blir 11,4. Et gjennomsnitt nær det teoretiske gjennomsnittet på 11,1 vil en normalt kun få hvis en kjører programmet mange ganger.

Oppgaver til Avsnitt 1.1.9

1. Kjør programmet i *Programkode* 1.1.9 *b*) flere ganger. Hvilke resultater får du?
2. Istedenfor å kalle `randPerm` fra *Programkode* 1.1.8 *e*) flere ganger, kan vi la *randPerm* fra *Programkode* 1.1.8 *f*) omstokke en eksisterende permutasjon på nytt og på nytt. Bruk den ideen i *Programkode* 1.1.9 *b*). Kall f.eks. *antallMaks* 10 ganger (bruk for-løkke) og summér verdiene som den returnerer. Avslutt med å skrive ut gjennomsnittet. Kjør så programmet for flere n-verdier enn $n = 100000$.

1.1.10 Måling av tidsforbruk

Metoden *currentTimeMillis()* fra *class System* måler tiden i millisekunder

I Java er det mulig å måle tidsforbruk. Klassen *System* i *java.lang* har `public static long` *currentTimeMillis*(). Den returnerer tiden i millisekunder eller egentlig antallet millisekunder som har gått siden 1. januar 1970. Vi kan lese av tiden før algoritmen starter og lese av på nytt når den er ferdig. Differansen forteller hvor mange millisekunder algoritmen brukte.

Vi må la en algoritme gå en stund for å sikre oss brukbare tidsavlesninger. Ellers kan unøyaktigheten i systemet gi et tidsforbruk på 0. Når vi skal testkjøre våre ulike *maks*-metoder bør vi derfor bruke en svært stor tabell. Eller en mindre tabell og isteden kjøre algoritmen mange ganger på samme tabell.

Vi bruker den klassen som ble laget i *Programkode* 1.1.9 *b)*, dvs. `class Program`. De tre versjonene av *maks*-metoden må legges inn, men for å kunne skille dem fra hverandre omdøper vi den første (*Programkode* 1.1.2) til *maks*1, den andre (*Programkode* 1.1.4) til *maks*2 og den tredje (*Programkode* 1.1.5) til *maks*3. De to første metodene har begge samme type *for*-løkke og selve løkken bruker like lang tid begge steder. Vi kan kalle det algoritmens *faste kostnader* (eng: overhead). En egen metode kan måle de faste kostnadene:

```
public static int kostnader(int[] a)  // legges i class Program
{
  int m = 0;
  for (int i = 1; i < a.length; i++) {}  // en tom blokk
  return m;
}

// main-metoden i class Program skal nå inneholde:
int n = 100_000, antall = 2_000; // tabellstørrelse og gjentagelser
long tid = 0;                    // for tidsmåling
int a[] = randPerm(n);           // en permutasjon av 1, . .  n

tid = System.currentTimeMillis();    // leser av klokken
for (int i = 0; i < antall; i++) kostnader(a);
tid = System.currentTimeMillis() - tid;    // medgått tid
System.out.println("Faste kostnader: " + tid + " millisek");

tid = System.currentTimeMillis();    // leser av klokken
for (int i = 0; i < antall; i++) maks1(a);  // Programkode 1.1.2
tid = System.currentTimeMillis() - tid;     // medgått tid
System.out.println("Maks1-metoden: " + tid + " millisek");

tid = System.currentTimeMillis();    // leser av klokken
for (int i = 0; i < antall; i++) maks2(a);  // Programkode 1.1.4
tid = System.currentTimeMillis() - tid;     // medgått tid
System.out.println("Maks2-metoden: " + tid + " millisek");

tid = System.currentTimeMillis();    // leser av klokken
for (int i = 0; i < antall; i++) maks3(a);  // Programkode 1.1.5
tid = System.currentTimeMillis() - tid;     // medgått tid
System.out.println("Maks3-metoden: " + tid + " millisek");
```

***Programkode* 1.1.10**

Resultatet av *Programkode* 1.1.10 er avhengig av hvilket miljø det kompileres og kjøres i. I et litt eldre 32-bits miljø (Java 6, Intel Pentium 4, 2.8 GHz og Windows XP) kom dette:

```
Faste kostnader: 235 millisek
Maks1-metoden:   562 millisek
Maks2-metoden:   469 millisek
Maks3-metoden:   375 millisek
```

Det ser ut som at maks3 er bedre enn maks2 som igjen er bedre enn maks1. I et nyere 64-bits miljø (Java 7, Intel Core i3, 3.10 GHz og Windows 7) kom dette:

```
Faste kostnader:   1 millisek
Maks1-metoden:   175 millisek
Maks2-metoden:   100 millisek
Maks3-metoden:    85 millisek
```

De forbedrede resultatene skyldes raskere prosessor og at kompilator og virtuell maskin (JVM) har blitt bedre. Men rekkefølgen på de tre versjonen er som før, men nå er maks3 kun litt bedre enn maks2. Med Java 8, men med samme konfigurasjon som rett over, ble resultatet dette:

```
Faste kostnader:   0 millisek
Maks1-metoden:   140 millisek
Maks2-metoden:    94 millisek
Maks3-metoden:    78 millisek
```

Det er forsatt en relativt stor forskjell mellom maks1 og maks2. Det betyr nok at å redusere antallet tabellaksesser (forskjellen mellom maks1 og maks2) fortsatt har betydning. Men den «klassiske» idéen med å legge inn en vaktpost (maks3) for å redusere arbeidet i en for-løkke, gir ikke så stor effekt. Men ting utvikler seg hele tiden. Her er en ny kjøring (Java 10, Intel Core i7-7700, 3.60 GHz og Windows 10):

```
Faste kostnader:   1 millisek
Maks1-metoden:    87 millisek
Maks2-metoden:    49 millisek
Maks3-metoden:    50 millisek
```

Her er den nyeste kjøringen med samme konfigurasjon som over, men nå med Java 11:

```
Faste kostnader:   2 millisek
Maks1-metoden:    85 millisek
Maks2-metoden:    42 millisek
Maks3-metoden:    38 millisek
```

Vi ser at det nå er liten forskjell på maks3 og maks2. Det har nok sammenheng med at kompilatoren ikke er istand til å optimalisere den «uortodokse» koden i maks3. Men fortsatt er maks2 bedre enn maks1.

Oppgaver til Avsnitt 1.1.10

1. Utvid *class Program* og gjør om *main*-metoden slik som det bes om i *Programkode* 1.1.10. Kjør programmet! Du kan endre tidsforbruket ved å endre tabellstørrelse og antall gjentagelser. Lag f.eks. tabellen være dobbelt så stor (n = 200_000).
2. Gjør om *main*-metoden slik at tabellen *a* inneholder tallene fra 1 til *n* i sortert rekkefølge. Hvordan går det da med tidsforbruket?
3. Har du tilgang til flere datamaskiner? Er tidsforbruket avhengig av hvilket miljø dette kompileres og kjøres i?

1.1.11 Maks og min i Java

Å finne den største (eller minste) verdien i en tabell er en såpass vanlig oppgave at Java burde ha en ferdig metode. Nå er det selvfølgelig svært enkelt å lage det selv, men det er som oftest fordelaktig å bruke en ferdig metode. Klassen Arrays i java.util er en klasse for tabellmetoder og den inneholder mange ulike, men dessverre ingen maks- eller min-metoder.

Men hvis vi isteden legger vår tabell inn i en Stream, så får vi tilgang til maks- og min-metoder og andre aktuelle metoder. Klassen Arrays har nemlig metoden stream(int[] a) som returnerer en IntStream. En stream er rett og slett en serie (en strøm) av verdier:

```
IntStream s = Arrays.stream(a);
```

Klassen IntStream inneholder mange forskjellige metoder, f.eks. *max* og *min*. De returnerer begge en OptionalInt (se også *Avsnitt* 1.1.7). Dette kan vi bruke slik:

```
import java.util.Arrays;
import java.util.stream.IntStream;
import java.util.OptionalInt;

public class Program
{
  public static void main(String ... args)
  {
    int[] a = {8,3,5,7,9,6,10,2,1,4};  // en heltallstabell
    IntStream s = Arrays.stream(a);    // fra int-tabell til IntStream

    OptionalInt resultat = s.max();    // kaller max-metoden

    if (resultat.isPresent()) System.out.println(resultat.getAsInt());
    else System.out.println("Ingen verdi!");
  }
}
```

Programkode 1.1.11

En Stream er en strøm av verdier. Når alle verdiene har passert, er strømmen tom. Det betyr at hvis vi i *Programkode* 1.1.11 også vil finne den minste verdien (metoden min()), må vi lage en ny strøm. Se oppgavene.

Oppgaver til Avsnitt 1.1.11

1. Kjør *Programkode* 1.1.11. Utvid slik at vi i tillegg får den minste verdien (metoden min()). Da må det lages en ny strøm. Hva skjer hvis tabellen *a* er tom (lengde lik 0)?
2. Metoden *sum*() finner summen som et int-tall. Bruk den til å finne tabellens sum. Hva skjer hvis du bruker en tom tabell?
3. Metoden *average*() returnerer en OptionalDouble og den har metoden *getAsDouble*(). Gjør om *Programkode* 1.1.11 slik at tabellens gjennomsnittsverdi skrives ut. Da må det øverst være med: `import java.util.OptionalDouble;`
4. Metoden *summaryStatistics*() returnerer en IntSummaryStatistics. Da får en både maks, min, gjennomsnitt, sum og antall på én gang. Prøv den.
5. Grensesnittet IntStream har mange metoder med *funksjonsgrensesnitt* som parameter. Klarer du å bruke noen av dem. Vi skal diskutere slike metoder senere.

1.1.12 Oppsummering

- En algoritme er en beskrivelse av de arbeidsoperasjonene som skal til for å løse en bestemt oppgave. Hvis det finnes flere algoritmer som løser den samme oppgaven, bør vi selvfølgelig velge den «beste». Hvilken som er best er situasjonsbestemt. Eksempel: Vi skal sortere en samling verdier. Da er kanskje én algoritme best hvis det er mange verdier, mens en annen er kanskje best hvis det er få verdier. Slike spørsmål vil vi diskutere mange ganger i dette faget.
- Når vi lager programkode for en algoritme, skal vi normalt bruke standard kodeteknikk og sørge for at koden blir klar, lett forståelig og godt lesbar. Da vil moderne kompilatorer ofte være i stand til å optimalisere koden for oss. Men vi bør selvfølgelig unngå å lage unødvendig ineffektiv kode. Spesielt skal vi være oppmerksomme på operasjoner som utføres mange ganger, dvs. operasjoner som inngår i løkker. Der er det viktig å unngå unødvendig arbeid.
- Et stort programsystem vil inneholde mange algoritmer. En erfaring er at enkelte deler av programsystemet utføres oftere enn de øvrige. Det kan være så skjevt at 80 prosent programtiden foregår i 20 prosent av koden. Derfor bør man under programmeringen bruke anerkjente teknikker og sørge for god programstruktur. Hvis det så under testingen skulle vise seg at programmet har «flaskehalser», så er det der en bør gå inn med «lure» teknikker for å øke effektiviteten.

I Java 8 er det mange nye klasser og nye konstruksjoner. En del av det vil bli tatt opp der det er aktuelt. En spesielt viktig nyhet er begrepet *lamda-uttrykk*. Det handler om hvordan en på en forholdsvis enkel måte kan ha funksjoner (eller metoder som det normalt heter i Java) som parametre til andre metoder. F.eks. har klassen `OptionalInt` flg. offentlige metoder:

```
public final class OptionalInt
{
  public static OptionalInt empty();
  public static OptionalInt of(int value);
  public int getAsInt();
  public boolean isPresent();
  public void ifPresent(IntConsumer consumer);
  public int orElse(int other);
  public int orElseGet(IntSupplier other);
  public boolean equals(Object obj);
  public int hashCode();
  public String toString();
}
```

For eksempel vil flg. kode kunne erstatte innholdet i main-metoden i *Programkode* 1.1.11:

```
int[] a = {8,3,5,7,9,6,10,2,1,4};
Arrays.stream(a).max().ifPresent(System.out::println);
```

1.1.13 Antallet tall som er større enn det største foran

I *Avsnitt* 1.1.6 står det som en påstand at hvis vi har n forskjellige tall i rekkefølge, vil det i gjennomsnitt være $H_n - 1$ av dem som er større enn det største av de foran. H_n er det n–te harmoniske tallet, dvs. summen av de inverse heltallene fra 1 til n:

1.1.13 *a)* $$H_n = 1 + 1/2 + 1/3 + 1/4 + \cdot\cdot\cdot\cdot + 1/n$$

Påstand: Av n forskjellige tall i rekkefølge, er det gjennomsnittlig $H_n - 1$ av dem som er større enn det største av de foran.

Påstanden kan reformuleres til at summen, over alle de $n!$ permutasjonene, av antallet tall som er større enn det største av de foran, er lik $n! \cdot (H_n - 1)$. Det spiller ingen rolle om vi opererer med n forskjellige tall eller tallene fra 1 til n.

1) Påstanden er sann for $n = 1$. Antallet tall som er større enn det største av de foran, er da lik 0 siden vi har kun ett tall. Men også $1! \cdot (H_1 - 1)$ er 0. I *Avsnitt* 1.1.6 tellet vi opp antallene og fant at påstanden også var sann for $n = 2$, 3 og 4.

2) Induksjonshypotesen: Anta at påstanden er sann for n.

Vi skal vise at påstanden er sann for $n + 1$. I en permutasjon er det et bestemt antall tall som er større enn det største av tallene foran. La $A(n + 1, k)$, $k = 1, \ . \ . \ , n + 1$ være summen av disse antallene for de $n!$ permutasjonene av tallene fra 1 til $n + 1$ der k står bakerst.

Anta først at $k \leq n$. Da gir induksjonshypotesen at $A(n + 1, k) = n! \cdot (H_n - 1)$ siden et tall mindre enn $n + 1$ bakerst ikke er større enn det største av de foran (som jo er $n + 1$). Hvis $k = n + 1$ får vi ett ekstra tall som er større enn det største foran. Induksjonshypotesen gir dermed at $A(n + 1, n + 1) = n! \cdot (H_n - 1) + n!$ Summerer vi dette for k lik 1 til $n + 1$ og bruker at $(n + 1)H_n + 1 = (n + 1)H_{n+1}$, får vi:

1.1.13 *b)* $$(n + 1) \cdot n! \cdot (H_n - 1) + n! = (n + 1)! \cdot (H_{n+1} - 1)$$

1.1.13 *b)* viser at påstanden er sann for $n + 1$. Ved hjelp av induksjonsprinsippet kan vi dermed si at påstanden er sann for alle $n \geq 1$.

Oppgaver til Avsnitt 1.1.13

1. Påstanden om at for n forskjellige tall i rekkefølge, vil gjennomsnittlig $H_n - 1$ av dem være større enn det største av de foran, kan vises på flere måter. La $B(n + 1 , k)$ være antallsummen for de $n!$ forskjellige permutasjonene av 1 til $n + 1$ der tallet $n + 1$ står fast på plass nr. $k + 1$, dvs. at det er nøyaktig k tall foran $n + 1$ (og dermed $n - k$ tall etter $n + 1$). Vis at $B(n + 1 , k) = n! \cdot H_k$. Summer så disse for k lik 1 til n og vis at det gir svaret $(n + 1)! \cdot (H_{n+1} - 1)$

> *«The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.»*
>
> Donald E. Knuth [1967]