Eksamen i

Algoritmer og Datastrukturer

IAI 21899

Høgskolen i Østfold
Avdeling for informatikk
og automatisering

Torsdag 30. november 2000, kl. 09.00 - 14.00

Hjelpemidler:
Alle trykte og skrevne hjelpemidler. Kalkulator.

Oppgavesettet består av i alt 12 sider, inkludert forside og vedlegg, og er delt i fem hoveddeler som kan løses uavhengig av hverandre.

Hver deloppgave er prosentvis vektet. Dette gjenspeiler ikke vanskelighetsgraden, men hvor mye hver enkelt oppgave kommer til å bidra til sluttkarakteren.

Hvis ikke annet er angitt, skal programkode skrives i Java, som pseudokode, eller som en kombinasjon. Koden bør være oversiktlig og godt kommentert. Du behøver ikke legge vekt på generelle prinsipper for tilgang til og beskyttelse av data, men kan anta at du har ubegrenset tilgang (friendly access) til alle data og metoder i alle klasseobjekter. Du behøver heller ikke legge vekt på å gjøre koden særskilt robust. Les oppgavetekstene nøye, planlegg arbeidet og disponer tiden før du begynner på besvarelsen.

Skriv tydelig!

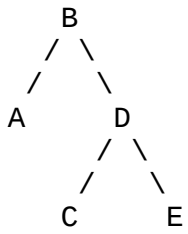
Lykke til!

Del 1: Binære søketrær – Totalt 20%

Tre nr. 1 og tre nr. 2 er binære søketrær.

Oppgave 1.1 - 5%

Tre nr. 1:



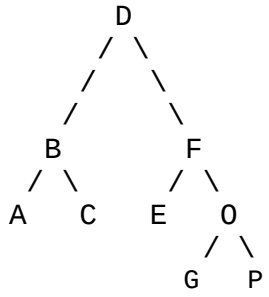
Sett inn en node med verdi 'F' i tre nr. 1.

Oppgave 1.2 - 5%

Anta at vi ønsker at tre nr. 1 skal være AVL-balansert etter innsetting av en ny node. Forklar hvorfor treet etter innsetting av 'F' (oppgave 1.1) ikke er AVL-balansert. Foreta en reparasjon av tre nr. 1 etter innsetting av 'F'. Tegn/kommenter hvert trinn i prosessen.

Oppgave 1.3 - 5%

Tre nr. 2:



Fjern noden med verdi 'D' i tre nr. 2.

Oppgave 1.4 - 5%

Anta at vi ønsker at tre nr. 2 skal være AVL-balansert etter fjerning av en node. Forklar hvorfor treet etter fjerning av 'D' (oppgave 1.3) ikke er AVL-balansert. Foreta en reparasjon av tre nr. 2 etter fjerning av 'D'. Tegn/kommenter hvert trinn i prosessen.

Slutt på del 1

Del 2: Sortering – Totalt 20%

En automatisk værstasjon måler lufttemperaturen. Hver måling er nummerert fortløpende etter hvert som de kommer inn, og blir lagret i en indeksert tabell. Måleren er følsom i intervallet –40 til +40 grader. Temperaturer under/over dette gir henholdsvis –40 og +40 grader målt verdi. I vedlegget er det

plottet utdrag av en slik serie målinger. I begge deloppgavene kan du anta at dataene du skal bearbeide får plass i internminnet.

Oppgave 2.1 - 10%

I denne oppgaven skal du lage en metode som sorterer tabeller som inneholder temperaturmålinger for hvert 10. sekund gjennom et helt år. Ved sorteringen skal du bruke en nøyaktighet på 0.01 grader, dvs. at temperaturene skal avrundes til 2 desimaler under sorteringen. Ved like temperaturer (etter avrunding) skal det sorteres på stigende løpenummer. Her er et utdrag fra en slik tabell:

Løpenr. (indeks)	42	43	44	45	46	47	48	49
Temperatur	2.1421	2.1343	2.1329	2.0299	2.1143	2.1239	2.0932	2.0722

Den sorterte sekvensen blir slik:

Løpenr.	45	49	48	46	47	43	44	42
Temperatur	2.03	2.07	2.09	2.11	2.12	2.13	2.13	2.14

Merk at 43 og 44 er like etter avrunding, og dermed kommer 43 før 44 selv om originalmålingene skulle tilsi det omvendte.

Implementer metoden ***`sorterAvrundet`*** *som er skissert i vedlegget. Metodens kompleksitet bør være lavere enn O(NlogN). Eksakt Javakode gir best uttelling. Angi kompleksiteten av metoden i O-notasjon.*

Oppgave 2.2 - 10%

Neste metode går ut på å sette inn måledataene fortløpende i en allerede sortert struktur. I dette tilfellet er det snakk om langt hyppigere målinger, ca. 10 i sekundet. Det blir derfor snakk om svært store datamengder, og metoden for å oppdatere en slik struktur (sette inn en ny måling) må være så rask som mulig. Denne gangen skal det sorteres på originalverdiene (altså *ikke* avrundet). Målingene blir foretatt med ”double” presisjon, og du kan anta at ingen målinger er nøyaktig like.

Drøft ulike kombinasjoner av datastrukturer og algoritmer som kan anvendes i dette tilfellet, med spesiell vekt på arbeidsmengde i O-notasjon.

Uansett valg av datastruktur/algoritme bør det være mulig å oppnå en effektivitetsgevinst ved å ta hensyn til tidligere innsettinger.

Lag en skisse (klartekst, ikke kode) av en mest mulig effektiv metode for innsetting av temperaturmålinger i en allerede sortert struktur. Beskriv både datastrukturen og algoritmen, og beregn arbeidsmengden i O-notasjon.

Slutt på del 2

Del 3, Binærheap – Totalt 30%

Oppgave 3.1 – 5%

Definisjonen av en binærheap innebærer bla.a. annet at den kan representeres som et *komplett* binærtre. Hvis vi nummerer nodene i et slikt tre ifølge levelorder, kan vi for enhver node beregne nummeret til foreldernoden og begge barna. Vi antar at nummeret på rota er *0* (null), og at node nr. *k* ikke er rota i treet og at den har to barn.

Hva er nummeret til foreldernoden til node k*?*
Hva er nummeret til venstre barn av node k*?*
Hva er nummeret til høyre barn av node k*?*

Oppgave 3.2 – 10%

Nedenfor finner du en binærheap representert i en tabell:

```
N E M T I R O G L A
```

Tegn det korresponderende binærtreet.

Oppgave 3.3 – 15%

I vedlegget finner du et utkast til klassen **`BinHeap`** som representerer en binærheap ved hjelp av et standard binærtre med referanser til venstre/høyre barn. Nodene i treet er av typen **`Node`**, som du også finner beskrevet i vedlegget. Klassen **`BinHeap`** inneholder en metode som bygger opp treet ut fra en heap gitt på tabellform (char[] tabell, jfr. forrige oppgave).

Implementer metoden **`lagTre`** *i klassen* **`BinHeap.`** *Metoden skal benytte den rekursive metoden* **`lagGrein`**, *som du også skal implementere. Eksakt kode gir best uttelling.*

Slutt på del 3

Del 4, Algoritmeanalyse – Totalt 20%

I denne delen skal vi se litt på to sorteringsalgoritmer, **`lrSort`** og **`rimiSort`**. Begge algoritmene sorterer en tabell av vilkårlige heltall i stigende rekkefølge.

I vedlegget finner du resultatene av å måle kjøretidene for algoritmene på tabeller av varierende størrelse med tilfeldig sammensatt innhold. Som en sammenlikning er det også tatt med måledata fra tilsvarende kjøringer med standard innsettingssortering.

Oppgave 4.1 – 10%

Bruk måledataene til å beregne kompleksiteten i O-notasjon på algoritmene.

Oppgave 4.2 – 10%

I vedlegget finner du implementasjonen av de to sorteringsalgoritmene, **`lrSort`** og **`rimiSort`**.

Forklar hvordan algoritmene virker, og beregn teoretisk kompleksitet i O-notasjon ut fra implementasjonen.

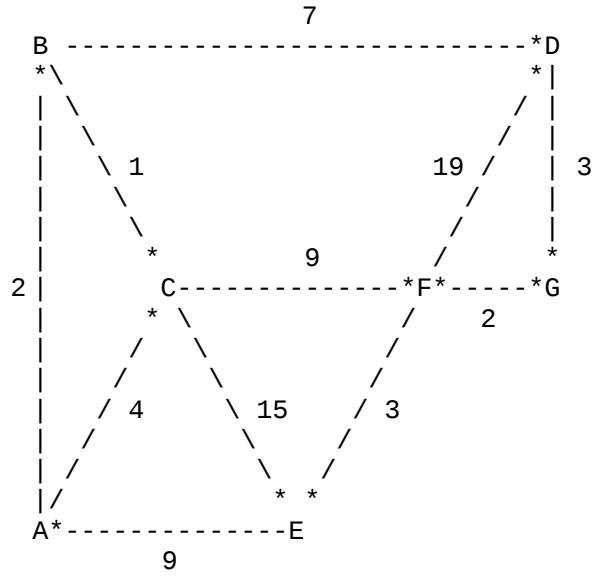
Kommenter forholdet mellom de praktiske testresultatene i oppgave 4.1 og dine teoretiske betraktninger.

Slutt på del 4

Del 5, Nettverk – Totalt 10%

Oppgave 5.1 - 10%

Under ser du en skjematisk framstilling av et rettet, vektet nettverk. En * i enden av en kant angir at retningen er *mot* stjernen. En stjerne i begge ender angir en dobbelt kant. Vektene er angitt som heltall. Nodene er merket med store bokstaver.



Anvend Dijkstras algoritme for å finne korteste vei fra node A til alle de andre nodene i nettverket. For hvert trinn i algoritmen skal du angi verdien på avstandsfunksjonen for hver node, nodens "forrige"-referanse, hvilken node som er under behandling, denne nodens naboer og nodene som er ferdigbehandlet.

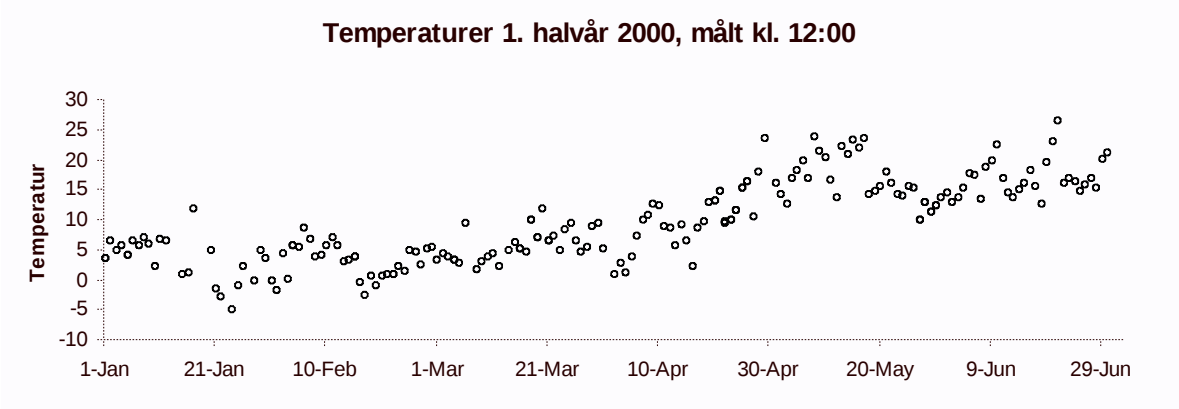
Skriv ut den korteste veien fra A til G.

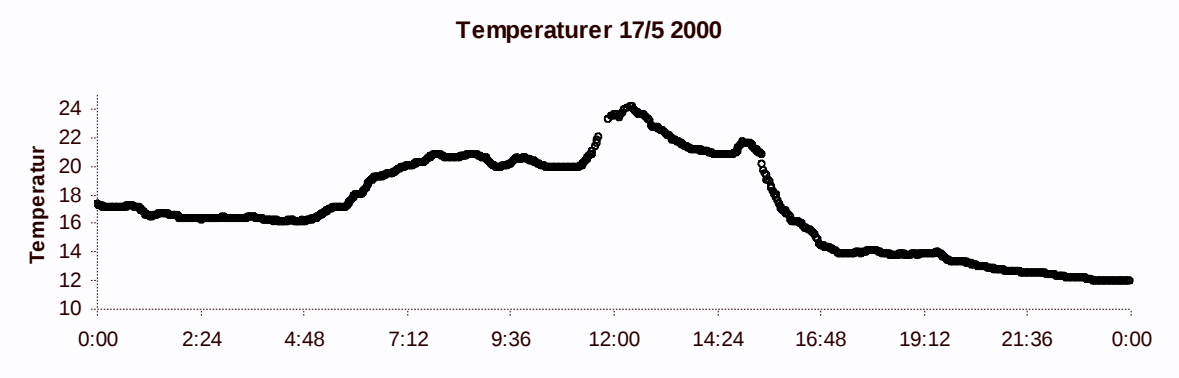
Slutt på del 5

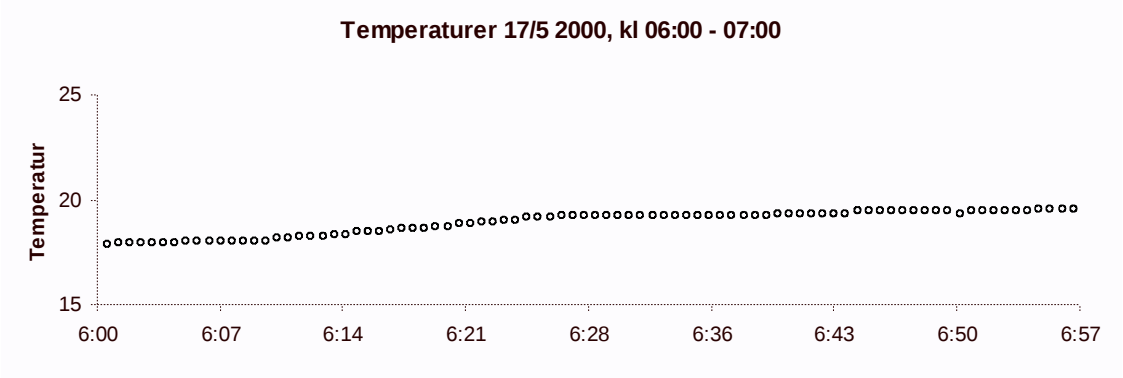
Vedlegg

Oppgave 2: Eksempel på værdata

Her har vi plottet temperaturer målt gjennom et halvt år, over et døgn og gjennom en time. Dette skal ikke brukes direkte i oppgaven, men er ment som en veiledende illustrasjon.







Oppgave 2: `class TempSort`

Klassen **`TempSort`** inneholder metoden **`sorterAvrundet`** som skal implementeres i oppgave 2.1. Du står fritt til å legge til flere variabler og/eller metoder i denne klassen**.**

```
class TempSort {

   // Metoden tar i mot en usortert tabell med double presisjon og
   // returnerer en float tabell med sorterte temperaturer slik
   // det er beskrevet i oppgaveteksten.

   float[] sorterAvrundet(double[] tab) {
      ...
   }
   ...

}
```

Oppgave 3: `class Binheap` og `class Node`

Klassene brukes til å representere en binærheap med nodereferanser. To av metodene skal implementeres ferdig i oppgave 3.3. Du står fritt til å legge til flere variabler og/eller metoder i klassen **`BinHeap.`**

```
class BinHeap {

   // Denne skal implementeres i oppgave 3.3
   // ved hjelp av den rekursive lagGrein
   void lagTre(char[] tre) {
      ...
   }

   // Rekursiv metode som bygger opp et subtre fra en tabell
   // Skal implementeres i oppgave 3.3

   Node lagGrein(/*Velg selv passende parametere*/) {
      ...
   }

   ...
   Node rot;
   ...

}
```

```
// Representasjon av Node
// Brukes i klassen BinHeap
// OBS: Ferdig implementert og klar til bruk

class Node {
  char verdi;

  Node(char c, Node v, Node h) {
    verdi = c;
    vBarn = v;
    hBarn = h;
  }

  Node vbarn;
  Node hBarn;
  char verdi;
}
```

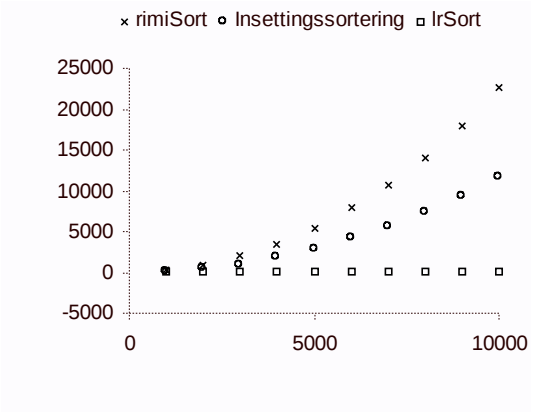
Oppgave 4: Tidsmålinger

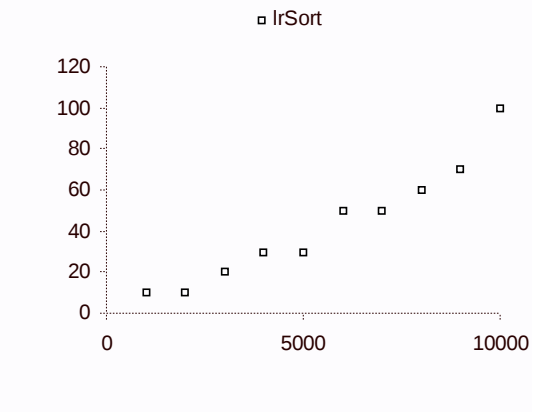
For hver av algoritmene lrSort, rimiSort og innsettingssortering har vi tatt tiden det har tatt å kjøre algoritmen på tilfeldig sammensatte tabeller av lengde N. For å få fram egenskapene til funksjonen T(N), dvs. tiden det tar å kjøre algoritmen for et gitt antall elementer N, har vi plottet på tre ulike måter (x-aksen representerer alltid antall elementer, N):

1. Med T(N) som verdier på y-aksen, dvs. de målte verdiene
2. Med T(N) / ($N*\log_2 N$) på y-aksen
3. Med T(N) / N^2 på y-aksen

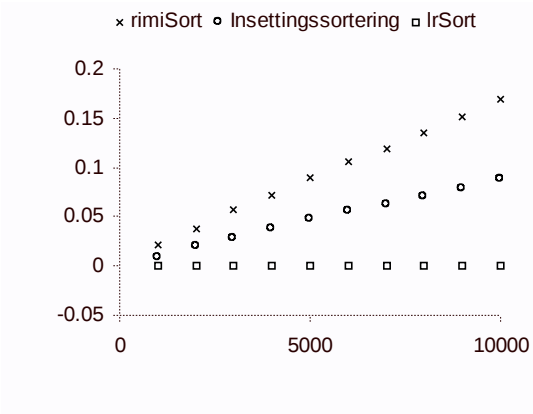
Det viser seg at lrSort er betydelig raskere enn både rimiSort og innsettingssortering. Derfor har vi for hver plottemetode i tillegg behandlet lrSort for seg selv.

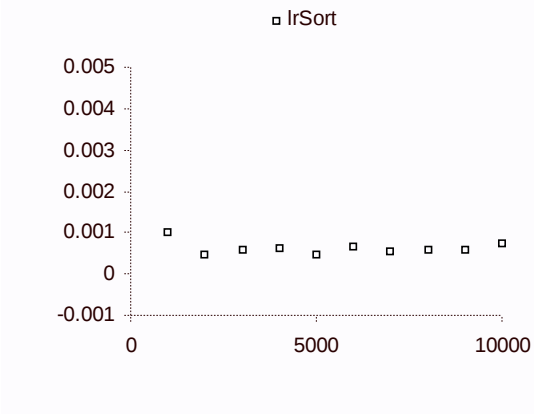
Y-AKSE: T(N), X-AKSE: N



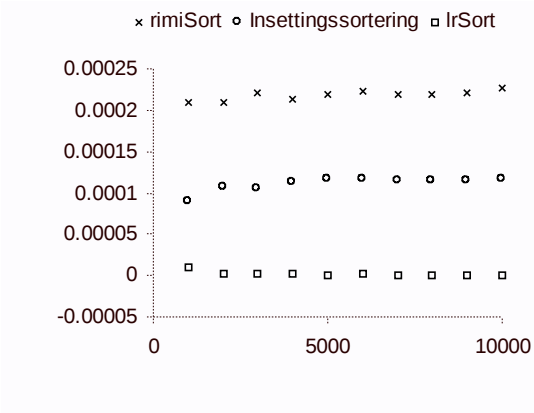


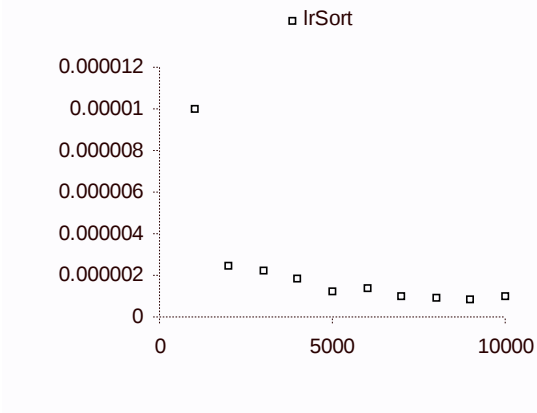
Y-AKSE: T(N) / (N*$\log_2$N) , X-AKSE: N





Y-AKSE: T(N) / N^2, X-AKSE: N





Oppgave 4: `class Sort`

Klassen implementerer to sorteringsalgoritmer som sorterer en heltallstabell i stigende rekkefølge.

```
class Sort {

  void lrSort (int[] tab) {

    for(int i = tab.length / 2; i >= 0; i--)  {
      flytt (tab, i, tab.length);
    }

    for(int i = tab.length - 1; i > 0; i--) {
      bytt   (tab, 0, i);
      flytt  (tab, 0, i);
    }
  }

  void flytt(int[] tab, int i, int n) {
    int  venstre;
    int  tmp;

    for(tmp = tab[i]; (2*i) + 1 < n; i = venstre) {
      venstre = (2*i) + 1;
      if(  venstre != n - 1 &&
           tab[venstre] < tab[venstre+1]) {
        venstre++;
      }
      if(tmp < tab[venstre]) {
        tab[i] = tab[venstre];
      }
      else
        break;
    }
    tab[i] = tmp;
  }
```

```
  void rimiSort (int[] tab) {
    hokuspokus(tab, 0, false);
  }

  void hokuspokus(int[] tab, int i, boolean slutten) {

    if (i ==  tab.length-1) {
      return;
    }

    if (!slutten) {
      hokuspokus (tab, i+1, false);
    }

    if (tab[i] > tab[i+1]) {
      bytt  (tab, i, i+1);
      hokuspokus(tab, i+1, true);
    }
  }

  // Felles metode som brukes i begge sorteringsalgoritmene
  void bytt(int[] tab, int i1, int i2){
    int tmp = tab[i1];
    tab[i1] = tab[i2];
    tab[i2] = tmp;
  }
}
```

Slutt på oppgavesettet