Tips til arbeidet med obligatorisk oppgave 2 i MAT-INF 1100 høsten 2004

Knut Mørken

3. november 2004

Etter samtale med noen av dere de siste dagene skjønner jeg at noen strever med del 2 av oblig2. Problemene ser både ut til å skyldes manglende oversikt over hva som skal gjøres, at noen av detaljene synes vanskelige, og diverse huller i programmeringskunnskapene. Målet med dette notatet er å gi en del informasjon som kan hjelpe til med å rette opp i dette. Mye av stoffet her pratet jeg også om på forelesningen mandag 1/11. For å ha særlig utbytte av dette notatet bør du først ha lest kapittel 10 i kompendiet. Føler du at du har god oversikt over problemet kan du sannsynligvis bare skumme eller hoppe over de første sidene og konsentrere deg om slutten av seksjon 1 og seksjon 2.

Her er det mest tips til oppgave 2a. Har du fått til 2a bør du kunne få til 2b med den informasjonen som allerede er gitt. For ordens skyld bør det nevnes at de som har fått til del 2 på egen hånd neppe vil ha særlig glede av å lese disse sidene.

1 Oversikt

En utfordring med programmering er at en samtidig både må ha oversikt og syn for detaljer. Som regel er eneste farbare vei å først skaffe seg oversikt før en graver seg ned i detaljene, og kompresjon av lyd er intet unntak fra denne regelen. I denne seksjonen skal vi se på problemet med lydkompresjon med fokus på overordnet struktur.

1.1 `MatInf1100Sound`

I INF 1000 har dere lært litt om objektorientering, og representasjon av lyd er en anvendelse som egner seg godt til å illustrere noen aspekter av dette

temaet. Programmene dere har fått utdelt for å manipulere lyd gjør derfor bruk av noen klasser. La oss først se litt på `MatInf1100Sound`.

Fra kompendiet vet vi at et digitalt lydsignal er en følge av tall som angir hvordan lufttrykket varierte ved mikrofonen da lyden ble tatt opp. Når vi spiller av lyden, setter vi høytalermembranen til å svinge på en tilsvarende måte. Disse svingningene brer seg gjennom lufta, får trommehinnen til å svinge på samme måte og dette oppfatter vi som lyd. Skulle vi hatt en eksakt representasjon av en lyd måtte vi kjent lyfttrykket med uendelig presisjon ved alle mulige tidspunkter. Dette lar seg selvsagt ikke gjøre, og når vi opererer med digital lyd måler vi lufttrykket et passende antall ganger pr. sekund med en passende nøyaktighet. Antall målinger pr. sekund kaller vi samplingsraten som vi betegner med r, og for CD-lyd vet vi at $r = 44100$. Dette betyr at tidsavstanden mellom målingene er $1/44100s \approx 0.000022s = 0.022$ms. Målingene gir opphav til et stort antall tall som følger etter hverandre i tid, og matematisk er det naturlig å tenke på dette som en (endelig) følge som vi kan kalle $\{a_i\}_{i=1}^N$, der N angir antall målinger. Lengden av lyden i sekunder ser vi da er gitt ved N/r. Siden $\{a_i\}_{i=1}^N$ er målt, er det naturlig å tenke på disse som desimaltall med et begrenset antall siffer. All informasjon om vårt digitale lydsignal er altså å finne i størrelsene r (samplingsraten) og følgen $\{a_i\}_{i=1}^N$ (samplene).

Hvis vi skal lagre et digitalt lydsignal på datamaskin er det nettopp disse størrelsene som må lagres, og siden de til sammen angir lyden er det fint å kunne knytte dem sammen når vi programmerer. Det er nettopp det vi oppnår ved å legge dem i en klasse. Hvis vi kaller lydklassen vår `MatInf1100Sound` er etter dette en naturlig definisjon

```
class MatInf1100Sound
{
    double sampleRate:
    short samples[];
}
```

Slik tradisjonen er har vi brukt litt mer forståelige variabelnavn enn det som er vanlig i matematikk. Samplingsraten r har vi lagt inn i `sampleRate` som er en `double`-variabel. Dette er naturlig siden samplingsraten ikke nødvendigvis er heltallig (men det hadde forsåvidt holdt å bruke en `float`-variabel). En array er jo ikke noe annet enn en endelig, ordnet sekvens av tall (en endelig følge) så det er naturlig at samplene $\{a_i)_{i=1}^N$ er lagret i en array, her kalt `samples`. At den skal være av type `short` er kanskje ikke så naturlig, men det er heller ikke umulig. Våre måleverdier vil typisk variere mellom noen ytterpunkter, og ved å multiplisere disse med passende verdier kan

vi alltid få skalert det hele over til en vilkårlig skala. Valget av `short` har ellers tekniske årsaker. Legg merke til at vi ikke har lagret antall sampler N. Dette er ikke nødvendig siden vi kan få tak i lengden til `samples` ved Java-operasjonen `samples.length` (men husk at Java indekserer arrayer fra 0 slik at a_1 er lagret i `samples[0]` mens a_N er lagret i `samples[N-1]`.

Hvis du går inn og titter på fila `MatInf1100Sound.java` så vil du se at definisjonen av klassen inneholder de to parametrene over pluss to andre som vi ikke skal komme inn på her.

I tillegg til å kunne lagre digital lyd ønsker vi også å kunne gjøre noen operasjoner på slike objekter, og ved hjelp av klassebegrepet kan vi koble disse metodene direkte mot `MatInf1100Sound`. Ser du på fila `MatInf1100Sound` vil du finne blant annet finne en metode som har første linje

```
MatInf1100Sound(short [] samples, double sampleRate)
```

Dette er en konstruktør som nettopp lager et lydobjekt fra samplene og samplingsraten. Ser du videre nedover fila vil du finne mange andre metoder, blant annet flere varianter av `play` som spiller av en `MatInf1100Sound` og en lang metode som danner en lyd av type `MatInf1100Sound` ved å lese data fra fil.

1.2 `Compressed1100Sound`

Det viser seg at det generelt er vanskelig å få til særlig kompresjon av lyd som er lagret som sampler og samplingsrate. Skriver vi derimot om lyden som $f = g + e$, slik som beskrevet i kompendiet, viser det seg at utsiktene til kompresjon øker. For å kunne programmere en slik kompresjon må vi finne en passende måte å representere g og e i et program. Siden g og e tilsammen representerer f og derfor hører sammen, er det fint å kunne referere til disse samlet. Igjen viser det seg at klassebegrepet kan hjelpe oss.

I kapittel 10 i kompendiet skrev vi stykkevis lineære funksjoner definert på intervallet $[a, b]$ på formen $f(x) = L_h(c_0, \ldots, c_{2n})(x)$. Her er hjørnene til f gitt ved punktene $\{x_i, c_i\}_{i=0}^{2n}$ der

$$x_i = a + ih, \quad \text{for } i = 0, \ldots, 2n$$

og $h = (b - a)/(2n)$. Når vi arbeider med lyd er det naturlig å sette $a = 0$ mens h er tidsavstanden mellom punktene, altså $1/r$, slik som nevnt over. Vi kan derfor lagre f ved å lagre $\{c_i\}_{i=0}^{2n}$ og $r = 1/h$, akkurat som vi gjør i `MatInf1100Sound`.

Når f skrives om som $f = g + e$ vet vi fra kompendiet at vi kan skrive g og e som

$$g(x) = L_{2h}(d_0, \ldots, d_n)(x), \tag{1}$$
$$e(x) = L_h(0, w_0, 0, w_1, \ldots, 0, w_{n-1}, 0)(x), \tag{2}$$

der

$$d_i = c_{2i}, \qquad \text{for } i = 0,\, 1,\, \ldots,\, n; \tag{3}$$
$$w_i = c_{2i+1} - \frac{d_i + d_{i+1}}{2}, \qquad \text{for } i = 0,\, 1,\, \ldots,\, n-1. \tag{4}$$

Fordelen med denne omskrivningen er at for et typisk lydsignal vil mange av w'ene bli små slik at de kan settes lik null uten at feilen blir så stor. Gjør vi dette blir e endret til $\tilde{e}$, og når vi lagrer g og $\tilde{e}$ på en fil vil den forhåpentligvis inneholde så mange nuller at et program som `gzip` klarer å komprimere den kraftig.

Vi vet hvordan vi kan representere f som et `MatInf1100Sound`-objekt, men hvordan kan vi representere g og e? (Når vi bare diskuterer representasjon kan vi glemme $\tilde{e}$ og konsentrere oss om e siden de begge er på formen (2)). Som nevnt over bør g og e representeres sammen siden de til sammen representerer f. I og for seg kan både g og e representeres som `MatInf1100`-objekter, men da må vi lagre alle nullene til e. Dette virker noe unødig, selv om `gzip` nok ville kunne kompensere godt for dette.

Det er alltid en fordel å gjøre ting enkelt, og vi ser at g og e er fullstendig gitt ved de to følgene $\{d_i\}_{i=0}^{n}$, $\{w_i\}_{i=0}^{n-1}$ og samplingsraten r (vi vet jo at $h = 1/r$ og at $2h = 1/(r/2)$). Hvis vi gir formatet vi lagrer g og e i navnet `Compressed1100Sound` er en naturlig definisjon derfor

```
class Compressed1100Sound
{
    double sampleRate;
    short d[];
    short w[]
    short eps;
}
```

I tillegg har vi her tatt med variabelen `eps` slik at vi kan vite hvor mye w'ene i verste fall er endret.

Det er viktig å være klar over at siden $f = g + e$, så kan vi representere f både som et `MatInf1100`-objekt og som et `Comressed1100Sound`-objekt. Vi bruker typisk `MatInf1100` ved lesing av lyd fra fil og ved avspilling, mens `Compressed1100Sound` altså egner godt om vi ønsker å kunne lagre lyden

kompakt på fil. For å kunne ha gleden av begge formatene må vi ha kode som konverterer begge veier, og det er disse metodene du skal programmere i del 2 av obligen.

Den første metoden vår konverterer et `MatInf1100Sound`-objekt til et `Compressed1100Sound`-objekt. Siden enhver klasse må ha en konstruktør som kan lage klasseobjekter, er det rimelig å la denne metoden være konstruktør for klassen `Compressed1100Sound` og være deklarert som

```
Compressed1100Sound(MatInf1100Sound f)
```

Hvis `lyd` er et `MatInf1100Sound`-objekt vil vi da kunne si

```
lyd_compressed = new Compressed1100Sound(lyd);
```

med det resultat at `lyd_compressed` vil inneholde en peker til et objekt som inneholder den opprinnelige samplingsraten til lyden og de to følgene $\{d_i\}$ og $\{w_i\}$ som beskriver g og e. Matematikken for å gjøre denne konverteringen finner du i Lemma 10.2.

Den andre metoden, som vi kan kalle `decompress`, går andre veien. Den begynner med g og e, representert som et `Compressed1100Sound`-objekt, og konverterer til et `MatInf1100Sound`-objekt, slik som beskrevet i Lemma 10.3. Det er rimelig å plassere også `decompress` i klassen `Compressed1100Sound` slik at vi har alle kompresjonsrelaterte metoder samlet på et sted. Men `decompress` kan ikke være en kontruktør siden den jo skal produsere et `MatInf1100Sound`-objekt. Deklarasjonen vil se ut som

```
public MatInf1100Sound decompress()
```

Denne metoden trenger ingen parametre siden den sitter inne i klassen og dermed har tilgang på attributtene til et klasseobjekt. Den er deklarert som `public` slik at den blir tilgjengelig overalt der klassen er tilgjengelig. Hvis vi nå har et objekt `lyd_compressed` av typen `Compressed1100Sound` og en variabel `lyd` av typen `MatInf1100Sound` kan vi si

```
lyd = lyd_compressed.decompress();
```

Resultatet er at den dekomponerte lyden i `lyd_compressed` blir konvertert til standard `MatInf1100Sound`-format i `lyd`, altså det motsatte av det som konstruktøren over gjør.

Grovstrukturen til klassen `Compressed1100Sound` er nå klar:

```
class Compressed1100Sound extends AbstractCompressedSound
{
    double sampleRate;
    short d[];
    short w[]
    short eps;

    Compressed1100Sound(MatInf1100Sound f)
    {
    }

    public MatInf1100Sound decompress()
    {
    }
}
```

Her har vi tatt med `extends AbstractCompressedSound` som har effekten at `Compressed1100Sound` i tillegg til attributtene vi har diskutert her også får med egenskapene til klassen `AbstractCompressedSound`. Dette er nødvendig hvis du vil benytte deg av testprogrammet som du finner på fila `CompressionTester.java`. Skriver du ditt eget testprogram kan du fjerne `extends AbstractCompressedSound`.

Vi kan nå fokusere på strukturen i de to metodene. `Compressed1100Sound` vil se ut omtrent som

```
Compressed1100Sound(MatInf1100Sound f)
{
    super(f);

    short c[];
    int clength, n;

    sampleRate = f.sampleRate;
    c = f.samples;
    clength = c.length;

    // Bestem n slik at 2n+1 = clength

    d = new short[n+1];
    // Beregn d som i formel 10.3
```

```
    w = new short[n];
    // Beregn w som i formel 10.4 og sett små w[i]'er lik 0
}
```

Denne beskrivelsen er riktig når antall sampler er et oddetall (lengden til `f.samples` er et oddetall). Den mystiske setningen `super(f)` kaller konstruktøren til `AbstractComressedSound`. Denne må stå aller først i konstruktøren, ellers trenger du ikke tenke mer over den (denne setningen var plassert feil i første utgave av `forslaga.java`). Sampleverdiene til `f` ligger jo i `f.samples`, og vi kunne godt ha referert til sampleverdiene på denne måten hele veien. Men for å få notasjonen nærmere kompendiet og forkorte skrivemåten, er det greit å la arrayen `c` peke på `f.samples`. Det er mindre å hente på å innføre variabelen `clength` siden vi bare sparer et punktum.

Det mest kritiske punktet er vel det å bestemme heltallet `n` slik at likheten `2n+1=clength` holder. I kompendiet har vi jo antatt at f har $2n+1$ sampler, så hvis vi kjenner n kan vi kode formlene 10.3 og 10.4 direkte.

I skissen over er det antatt at lengden til `f.samples` er et oddetall. Om dette ikke skulle være tilfelle, må vi gjøre en liten justering, se under.

Rekonstruskjonsmetoden `decompress` vil se ut som

```
decompress()
{
    int clength // Lengden til c som vi nå skal regne ut
                // kan vi finne fra lengden til w og d,
                // se lemma 10.3

    int n = w.length; // Kan være nyttig for å indeksere løkker

    short c [] = new short[clength]; // Setter av plass til
                                     // samplene til den
                                     // rekonstruerte lyden

    // Beregn c som i formelene 10.5 og 10.6

    return new MatInf1100Sound(c,sampleRate);
}
```

Litt avhengig av hvordan du kompenserer for partallslengde i konstruktøren vil du kanskje også måtte justere i `decompress`, se under.

1.3 Dekomposisjon over flere nivåer

For å få kraftig kompresjon må vi dekomponere flere ganger. Generelt kan vi skrive f som $f = g_k + e_k + \cdots + e_{k-1}$. For å programmere dette må vi tilpasse datastrukturen vår slik at vi kan representere flere feilfunksjoner. Java har en mekanisme som passer som hånd i hanske til dette, nemlig todimensjonale arrayer. Vi lar `w[0]`, `w[1]`, ..., `w[k-1]` angi koeffisientene til e_1, e_2, ..., e_k. Vi kan da referere til den i'te w'en i e_{j+1} som `w[j][i]`. Det fine med dette er at `w[j]` er en helt vanlig array slik at vi kan få tak i lengden til `w[j]` ved å si `w[j].length`. Vi oppnår dette ved å endre `Compressed1100Sound` ørlite grann til

```
class Compressed1100Sound extends AbstractCompressedSound
{
    double sampleRate;
    int k;
    short d[];
    short w[][];
    short eps;

    Compressed1100Sound(MatInf1100Sound f)
    {
    }

    public MatInf1100Sound decompress()
    {
    }
}
```

I konstruktøren må vi aller først bestemme en verdi for k. Det enkleste er å sette k til en fast verdi. En litt mer raffinert løsning er å la k være slik at g_k for eksempel inneholder omtrent 100 sampler. Etter at k er bestemt kan vi regne ut e_1, ..., e_k og g_k. For å gjøre dette er det naturlig å følge samme oppskrift som i kapittel 10. Vi løper i løkke og begynner med å dekomponere f som $f = g_1 + e_1$, deretter g_1 som $g_1 = g_2 + e_2$ også videre, inntil vi siste gang dekomponerer g_{k-1} som $g_{k-1} = g_k + e_k$. En skisse av konstruktøren kan se ut som

```
Compressed1100Sound(MatInf1100Sound f)
{
    super(f);
```

```
    short c[];
    int j, n;

    sampleRate = f.sampleRate;
    k = // Regn ut en passende verdi for k
    d = f.samples;

    for (j=0; j<k; j++)
    {
        c = d; // Når vi kommer hit vil d inneholde samplene
               // til g_j (vi kan tenke på f som g_0). Vi
               // skal nå regne ut en ny d og det er
               // da rimelig å la c betegne samplene til
               // g_j som nå skal dekomponeres

        // Bestem n slik at 2n+1 = c.length

        d = new short[n+1];
        // Beregn d som i formel 10.3

        w[j] = new short[n];
        // Beregn w[j] som i formel 10.4 og
        // sett små w[j][i]'er lik 0
    }
}
```

Som i oppgave 2a må vi justere algoritmen slik at den også håndterer tilfellet der lengden til `c` er et partall. I tillegg må metoden `decompress` justeres til å rekonstruere k ganger .

2 Testing

Som nevnt andre steder trenger du ikke skrive noe testprogram selv, du kan bruke programmet `CompressionTester.java` (det fins jo ingen `main()` i `Compressed1100Sound`). Pass på at versjonen av `Compressed1100Sound` som du ønsker å teste ligger på samme filområde som `CompressionTester`. Hvis du sier

```
javac CompressionTester.java
java CompressionTester
```

vil programmene dine bli kompilert og `CompressionTester` utført. Dette programmet genererer en standard lyd på `MatInf1100Sound`-format og lagrer denne på fila `original.obj`, kaller konstruktøren i `Compressed1100Sound`, lagrer den dekomponerte lyden på fila `Compressed1100Sound.obj`, rekonstruerer denne lyden ved hjelp av din `decompress` og spiller av den rekonstruerte lyden. Du kan se størrelsen på de to `obj`-filene ved å si

```
ls -l *.obj
```

og kode den siste kompakt ved å si

```
gzip Compressed1100Sound.obj
```

3 Detaljer

Her har jeg samlet noen tips som går mer på detaljer i programmeringen av del 2 av oblig2.

3.1 Hvordan oppdage at et tall er et partall?

En forutsetning for beskrivelsen i Lemma 10.2 er at antallet sampler i f er et oddetall. Hvis dette ikke er tilfelle må dekomposisjonsalgoritmen justeres. Vi skal se på hvordan algoritmen kan justeres under, men aller først må vi vite hvordan vi kan oppdage at et tall er et partall.

I tillegg til de vanlige aritmetiske operasjonene `+,-,*,/` har Java også en operasjon som betegnes med `%`. Denne angir rest ved heltallsdivisjon. Hvis `m` og `i` begge er av type `int` (eller `long`) vil derfor uttrykket `m % i` gi resten når `m` divideres med `i`. For eksempel er vil `5 % 3` gi resultatet 2 og `17 % 2` gi resultatet 1. Ved hjelp av `%` burde det derfor være ganske greit å sjekke om lengden til `c` er et partall.

Det er også mulig å sjekke dette uten `%`-operatoren siden det er lett å se at `(m/i)*i` er det samme som `m % i` (når `m` og `i` er heltall).

3.2 Lengden til `f.samples` er et partall

Hvordan bør dekomposisjonen endres om antal sampler er et partall? En strategi er foreslått i seksjon 10.3.4. Denne kan virke noe komplisert, men er kanskje den mest tilfredstillende siden den ekstra `w`'en regnes ut som en differense som det er godt håp om at kan være liten og som derfor kanskje kan bli null.

På den annen side skal vi huske på at en lyd typisk består av flere hundretusen sampler, så en 0 til eller fra i `w` betyr veldig lite. På denne bakgrunnen har vi minst to andre muligheter:

1. Erstatt `c` med en ny sample-array som består av samplene i `c`, med det siste samplet lagt til på slutten en ekstra gang.

2. Lat som lengden av `c` er én mindre enn den virkelig er. Dette betyr bare at det siste samplet blir ignorert slik at den dekomponerte lyden blir et sample kortere enn den gitte lyden.

Begge de to nummererte forslagene har som konsekvens at det å først gjennomføre en dekomposisjon med `eps=0`, etterfulgt av en rekonstruksjon, ikke tar oss tilbake til utgangspunktet, men det går det alså an å leve med siden endringene er så små.

Den løsningen som er enklest å programmere er nok den siste. Her er alt som trengs å gjøre å redusere med én, variabelen som angir lengden til `c`, før dekomposisjonen. Den justerte versjonen av konstruktøren vil da se ut som

```
Compressed1100Sound(MatInf1100Sound f)
{
    super(f);

    short c[];
    int clength, n;

    sampleRate = f.sampleRate;
    c = f.samples;
    clength = c.length;
    // Juster clength om den er et partall

    // Bestem n slik at 2n+1 = clength

    d = new short[n+1];
    // Beregn d som i formel 10.3

    w = new short[n];
    // Beregn w som i formel 10.4 og sett små w[i]'er lik 0

}
```