Øving 12 algoritmer og datastrukturer

Innhold

Øving 12 algoritmer og datastrukturer **1**
Innledning 1
Testfiler for komprimering 1
Krav til løsningen 2
Deloppgave Lempel-Ziv 2
Tips om Lempel-ziv 2
Deloppgave Huffmankoding 3
Tips om Huffmankoding 3
Javatips for begge deloppgaver 4
Noen kodeeksempler 5

Innledning

Lag et program som kan lese en fil og lage en komprimert utgave. Lag et annet program som pakker ut igjen (dekomprimerer) og gjenskaper originalen.

Bruk enten **Huffmankoding** eller **Lempel-Ziv** for å komprimere. Ressurssterke grupper med mange gode programmerere må gjerne lage begge deler for å oppnå best mulig kompresjon. I så fall anbefaler jeg å lage separate programmer, det gjør det enklere å teste og dele på arbeidet. Da går an å sjekke hvilken algoritme som komprimerer best, og om det er en fordel å komprimere output fra Lempel-Ziv med Huffmann.

Programmer som «zip» bruker Lempel-Ziv-Welsh for å komprimere, og deretter Huffmannkoding på output fra Lempel-Ziv.

Det kan bli en del arbeid, da håndtering av bits & bytes er nytt for mange. Det er derfor denne øvingen teller litt mer.

Testfiler for komprimering

Oppgavetekst (pdf)	`http://www.iie.ntnu.no/fag/_alg/kompr/opg12.pdf`
Oppgavetekst (txt)	`http://www.iie.ntnu.no/fag/_alg/kompr/opg12.txt`
Forelesningen (pdf)	`http://www.iie.ntnu.no/fag/_alg/kompr/diverse.pdf`
Forelesningen (txt)	`http://www.iie.ntnu.no/fag/_alg/kompr/diverse.txt`
Forelesningen (lyx)	`http://www.iie.ntnu.no/fag/_alg/kompr/diverse.lyx`

Krav til løsningen

1. Implementer enten Lempel-Ziv eller Huffmannkoding. (Eller begge deler, om dere har tid!) Andre algoritmer blir ikke godkjent medmindre det er avtalt på forhånd. Lempel-Ziv-Welsh (LZW) er en *annen* algoritme.
2. Dere må lage programmene selv, ikke noe «cut & paste» fra nettet. Grupper som ikke kan forklare detaljer i programmet sitt, får ikke godkjent denne oppgaven. Det er mye å lære av å gjøre en slik oppgave, som en ikke får med seg med «cut & paste». Både når det gjelder algoritmene, og generell programmering.
3. Komprimering og utpakking *skal* skje i separate kjøringer. Det er *ikke* greit å ha ett samleprogram som både gjør innpakking og utpakking i en operasjon. Utpakking skal *bare* trenge den komprimerte fila, ikke noen variabler/datastrukturer fra innpakkinga.
4. Programmene må lese og skrive *filer*. Altså ikke bare testdata i en tabell.
5. Utpakkingsprogrammet må produsere en fil som er identisk med originalen. Men det skal ikke trenge tilgang på originalfilen, *bare* den komprimerte filen.

   Likhet kan testes med «diff» (linux) eller «fc» (windows)
6. Komprimering må klare å spare minst 10% i forhold til originalen, for én av mine testfiler. Operativsystemet kan fortelle hvor store filene er, i bytes.
7. Programmet bruker ikke hasmap/hashset e.l., som ikke er nødvendig her.

Deloppgave Lempel-Ziv

Implementer en variant av Lempel-Ziv datakompresjon. (Men ikke Lempel-Ziv-Welsh)

Finn ut hvor mye programmet deres komprimerer testfilene mine. Det er ikke sikkert alle filtyper lar seg komprimere. Men for å få godkjent, må gruppa i det minste kunne komprimere en fil så den sparer 10%, og deretter pakke den ut igjen.

Gruppa må dessuten kunne forklare detaljene i programmene sine.

Tips om Lempel-ziv

Normalt blir det veldig lite kompresjon på små filer. Bittesmå filer kan brukes for å finne feil i programmet, men for å teste kompresjon bør filene minst være på noen kilobyte.

Det blir noen avgjørelser å ta, som f.eks. hvor langt bakover programmet deres skal lete etter repeterte sekvenser. Zip leter 32kB bakover, det fins også versjoner som går 64kB tilbake. Hvis dere lar programmet gå lenger tilbake, vil det bli tregere men sannsynligvis komprimere bedre også.

Om en vil ha et veldig kjapt program, kan det lønne seg å la seg inspirere av avanserte tekstsøkalgoritmer.

Filformat

Filformat bestemmer dere selv. Det kan fort bli en avveiing mellom hvor komplisert programmet skal være, og hvor godt det skal komprimere.

Den komprimerte fila kan bestå av blokker. Hver blokk starter med en byte-verdi, som er et tall mellom -128 og +127. Hvis tallet er negativt, f.eks. -57, betyr det at det er en serie med tegn som ikke lot seg komprimere. (I dette eksempelet, 57 tegn).

Hvis tallet er positivt, angir det lengden på en repetert sekvens. De neste 1, 2 eller 4 byte er et heltall som forteller hvor langt bakover i fila denne sekvensen er å finne. Med 1 byte (byte) er det bare mulig å gå 127 tegn bakover. Programmet blir raskt, men komprimerer antagelig ikke så kraftig. Med 2 byte (short) går det an å gå opp til 32 kB bakover, men vi bruker altså opp en ekstra byte. Med 4 byte (int) kan vi gå opp til 2 GB bakover. Det gir mange flere muligheter for å finne repeterte strenger, men bruker også mer plass. Et program som leter opptil 2 GB bakover, blir sannsynligvis temmelig tregt også. Det kan lønne seg å begrense litt...

Deloppgave Huffmankoding

Lag et program som leser inn en fil og genererer en huffmanntre ut fra byte-verdiene i filen. Deretter bruker programmet huffmanntreet til å skrive en komprimert huffmannkodet fil. Sjekk hvor mye plass dere sparer, ved å komprimere testfilene mine. Dere må også kunne pakke filene ut igjen.

For pakke ut, trenger utpakkingsprogrammet nok informasjon til å gjenskape huffmantreet. Det enkleste er å legge frekvenstabellen først i den komprimerte fila. Adaptiv huffmankoding er en mer avansert og krevende løsning.

For å få godkjent, må ihvertfall en av filene komprimeres med minst 10%.

Tips om Huffmankoding

Huffmanndata som trengs for å pakke ut igjen

Det er ikke nødvendig å lagre huffmanntreet, det holder å lagre frekvenstabellen. Utpakkingsprogrammet kan dermed bygge opp samme tre ut fra frekvensene.

```
int frekvenser[256];
```

En slik frekvenstabell blir alltid 1 kB, filen som skal komprimeres må dermed være stor nok til at komprimeringen sparer mer enn 1 kB.

Adaptiv Huffmannkoding

Med adaptiv huffmannkoding slipper man å lagre frekvensene også. Man deler fila opp i blokker med fast størrelse. Første blokk komprimerer man ikke, den bare kopieres til output. Samtidig lager man et huffmanntre. Neste blokk komprimeres med huffmanntreet fra forrige blokk. Samtidig oppdaterer man frekvensene, og lager nytt huffmanntre som brukes for neste blokk osv.

Adaptiv huffmankoding blir bedre, fordi den klarer å ta hensyn til at bokstavfordelingen endrer seg underveis.

Om bitstrenger

En bitstreng er *ikke* en streng som dette: "00001101". Dette er en *tekststreng* med 8 tegn. Skriver vi dette til en fil, går det med 8 byte, og vi oppnår ikke noe datakompresjon. Tvert imot får vi en veldig stor fil!

Men bitstrengen 0b00001101 er det samme som tallet 13, og kan lagres som én byte.

Datatypen «long» er på 64 bit. Ingen tegn vil trenge lenger Huffmankode enn det. (Det kan vises at nå man komprimerer en fil på 2.7GB, trenger ingen tegn kodes med mer enn 44 bit.) «long» er dermed egnet til å lagre bitstrenger. En «long» har alltid 64 bit, så en bitstreng-klasse må også ha et felt som forteller hvor mange av bitene som er med i bitstrengen.

Å skrive bitstrenger til fil, blir en del ekstra arbeid. Java lar oss bare skrive hele byte, og for å være effektive bør vi bare skrive byte-array av en viss størrelse. Men, med høyre/venstreskift samt binære & og | -operasjoner, kan vi få våre bitstrenger inn i et byte-array som så kan skrives til disk.

Tilsvarende for lesing: Vi leser inn et byte-array, og plukker deretter ut én og én bit for å navigere gjennom huffmanntreet.

Om koking

På nettet fins mange implementasjoner av Huffmannkoding. De har sine særegenheter som vi kjenner igjen. Programmer som bruker hashset/hasmap vil bli underkjent som kok. hashopplegg trengs ikke for å løse denne oppgaven.

Javatips for begge deloppgaver

Datatype	bits	byte	min	max
byte	8	1	-128	127
short	16	2	-32 768	32 767
char	16	2	0	65 535
int	32	4	-2147483648	2147483647
long	64	8	-9223372036854775808	9223372036854775807

Programmer som leser én og én byte fra fil, blir alltid trege i Java. For å få noe fart i sakene, lønner det seg å lese/skrive større blokker, f.eks. et array med bytes.

Jeg godkjenner imidlertid løsninger som leser/skriver én og én byte også – så lenge de ikke er for trege til å demonstreres. Noe bitfikling blir det uansett med Huffmannoppgaven. Det går ikke an å skrive «en halv byte» til fil, man må i det minste samle opp bits til man har en hel byte. Det kan være lurt å lage en egen klasse for å sende bitstrenger til fil.

Noen kodeeksempler

```
//Åpne filer:
innfil = new DataInputStream(new BufferedInputStream(new FileInputStream(inn_navn)));
utfil = new DataOutputStream(new BufferedOutputStream(new FileOutputStream(ut_navn)));

//Lese data fra fil inn i byte-array:
//   byte []data  : arrayet vi leser inn i
//   int posisjon : index i byte-array for det vi leser inn
//   int mengde   : antall byte vi vil lese inn
innfil.readFully(data, posisjon, mengde);

//Lese inn én byte
byte x;
x = innfil.readByte();
//Har også:
short s = innfil.readShort();
char c = innfil.readChar();
int i = innfil.readInt();
long l = innfil.readLong();

//Skrive data fra byte-array til fil:
utfil.write(data, posisjon, mengde);

//Skrive én byte til fil:
byte singlebyte = 17;
utfil.writeByte(singlebyte);
//Har også:
//utfil.writeChar(char c);
//utfil.writeShort(short s);
//utfil.writeInt(int i);
//utfil.writeLong(long l);

//Hente 13 bit fra long1, 8 bit fra long2 og 4 bit fra long3,
//og få det inn i et byte-array:

byte[] data = new byte[3];
long long1 = 0b1101000010011; //13 bit
long long2 = 0b11100111;      //8 bit
long long3 = 0b010;           //3 bit

//8 første bit fra long1 til data[0]
//øvrige bits maskeres bort med &
data[0] = (byte)(long1 & 0b11111111);

//5 gjenværende bit fra long1 til data[1]
//høyreskiftet fjerner bits vi allerede har lagt i data[0]
//trenger ikke maskere fordi resterende bits i long1 er 0.
data[1] = (byte)(long1 >> 8);

//data[1] har plass til 3 av de 8 bit fra long2
//venstreskifter 5 plasser fordi de 5 første bits i data[1] er i bruk fra før
//trenger ikke maskere vekk bits fordi bits over 256 ikke går inn i en byte uansett
data[1] |= (byte)(long2 << 5);

//5 gjenværende bit fra long2 til data[2]
//høyreskift fjerner de bits vi allerede la i data[1]
data[2] = (byte)(long2 >> 3);

//data[2] har plass til de 3 bit fra long3
data[2] |= (byte)(long3 << 5);
System.out.printf("%x %x %x\n", data[0], data[1], data[2]);
```