Øving 8 algoritmer og datastrukturer

Innhold

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 både **Lempel-Ziv** og **Huffmankoding** for å komprimere. LZ gjør om fila til en blanding av bakoverreferanser og ukomprimerte strenger. Innholdet i de ukomprimerte strengene komprimeres videre med Huffmannkoding.

Det kan bli en del arbeid, da håndtering av bits & bytes er nytt for mange. Det er derfor denne øvingen teller litt mer. Jobb gjerne i grupper, og fordel arbeidet. Les hele oppgaveteksten før dere begynner, unngå misforståelser.

Testfiler for komprimering

Oppgavetekst (pdf)	`http://www.iie.ntnu.no/fag/_alg/kompr/opg8-2021.pdf`
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`
100MB enwik8	`http://www.iie.ntnu.no/fag/_alg/kompr/enwik8`

enwik8 er en diger fil, de første 100MB av engelsk wikipedia. Dere trenger ikke håndtere den, men har dere et kjapt program, er det jo artig å prøve. Denne fila brukes i rekordforsøk, der

de beste har klart å presse den sammen til ca. 15MB. Det er pengepremier for de som klarer bedre…

Krav til løsningen

1. Send inn programmer for komprimering og dekomprimering, som klarer å komprimere og pakke ut igjen ihvertfall to av testfilene, og andre testfiler på samme størrelse. Send med en skjermdump som viser filstørrelser for originalfiler, komprimerte filer og utpakka filer, i fall jeg får uventede problemer med å kjøre programmene.
2. Komprimeringsprogrammet bruker både Lempel-Ziv og Huffmannkoding, for å lage minst mulig fil.
3. Den komprimerte fila må være *mindre enn originalen.* Dette måles ved å se på filstørrelser.
4. Dekomprimeringsprogrammet må greie å gjenskape originalfila, når det bare har den komprimerte fila å pakke ut.
   a) Ingen ekstra filer, alt som trengs må finnes i den komprimerte fila. Dekompresjon må altså virke uten å ha originalen tilgjengelig.
   b) At utpakket fil er helt lik originalen, kan f.eks. testes med «fc» i windows, eller «diff» på linux/mac.
5. Dere må lage programmene selv, ikke noe «cut & paste» fra nettet. 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.
6. 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.

   Dere *kan* ha ett stort program som gjør hele jobben, men det må i så fall være i to separate kjøringer. Et slikt program må altså stoppe etter komprimering, og kjøres igang med andre parametre for å pakke ut.
7. Programmene må lese og skrive *filer*. Altså ikke bare testdata i en tabell. Filene ligger i den mappa programmet kjører i, eller i undermappa «src/». Ingen annen mappestruktur!
8. Programmet bruker *ikke* Hasmap, Hashset eller andre former for hashtabeller.

Tips for best kompresjon

Det enkleste er å gjøre Huffman- og LZ-delene helt separat. Man kan man fordele arbeidet på gruppa på denne måten, og sette sammen til slutt. Debuggingen blir også enklere. Men det er ikke dette som gir best kompresjon.

Lempel-Ziv lager koder for repeterte strenger, samt tekststrenger som ikke lot seg komprimere. Strengene som ikke lot seg komprimere med LZ, kan Huffman-delen ta seg av. Huffmankompresjon fungerer vanligvis ikke så godt på kodene for repeterte strenger, så et smart Huffmanprogram kan skrive disse delene som de er, uten å forsøke å komprimere dem. Et slikt opplegg blir som regel bedre, men er mer arbeid å kode.

Tips om hele oppgaven

Det er lett å gjøre feil, og vanskelig å debugge. Når noe går galt, bruk testfiler som er så små at dere kan følge med på alt som skjer hele veien. F.eks. en testfil med bare 3–20 byte. Da blir det lettere å se når noe går galt. En måte er å bruke en debugger som kan vise innholdet i variabler. En annen måte er å legge inn debug-utskrifter for å se hva som skjer.

Det er lurt å teste og lage LZ-delen og Huffmann-delen hver for seg. Programmer som zip gjør alt i memory for å være effektive. Men det er ikke noe galt i å ha ett program som gjør LZ-komprimeringen og skriver sitt resultat til fil, og deretter et Huffmann-program som leser denne fila og skriver en ny og mer komprimert fil. Dekomprimering kan håndteres tilsvarende. Da kan hvert trinn testes og debugges for seg. Dette gjør det også lettere å dele på arbeidet i gruppa.

Deloppgave Lempel-Ziv

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

Tips om Lempel-ziv

Normalt blir det veldig lite kompresjon på små filer. Bittesmå filer kan brukes for å debugge programmet, men for å teste kompresjon bør filene være noen titalls 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 tekst-søkalgoritmer.

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 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, teller frekvenser og genererer en huffmanntre ut fra byte-verdiene i filen. Deretter bruker programmet huffmanntreet til å skrive en komprimert huffmann-kodet fil.

For å 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.

Tips om Huffmankoding

Huffmanndata som trengs for å pakke ut igjen

Det er ikke nødvendig å lagre huffmanntreet, det holder å lagre frekvenstabellen. Utpakkings-programmet 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. Filene jeg lenker til, skulle være store nok.

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. Men det er ikke et krav, poenget er kompresjon, ikke hastighet.

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]);
```