

Øving 8 algoritmer og datastrukturer

Innhold

Øving 8 algoritmer og datastrukturer	1
Innledning	1
Testfiler for komprimering	2
Krav til løsningen	2
Tips for best kompresjon	3
Tips om hele oppgaven	3
Tips om Lempel-Ziv	3
Filer	3
Litt om å kombinere LZ og Huffman	4
Litt om Lempel-Ziv-Welsh	4
Kombinere LZW og Huffman	5
Om Huffmankoding	5
Tips om Huffmankoding	5
Delvis uttelling	6
Javatips for begge deloppgaver	6
Noen kodeeksempler, bitoperasjoner og filbehandling	7

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 gjerne 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. Håndtering av bits & bytes er nytt for mange. Jobb gjerne i grupper, og fordel arbeidet. *Les hele oppgaveteksten før dere begynner, unngå misforståelser!*

Det er anledning til å jobbe i gruppe her, selv om man har jobbet alene på mindre øvinger.

Jeg kommer til å lage en highscore-liste sortert på best kompresjon av «diverse.lyx». For å komme på lista, må utpakking fungere feilfritt. Den halvparten som ligger øverst på lista når fristen er ute, får et ekstra poeng på oppgaven. Så her har dere et valg: dere kan gjøre dette til en liten enkel oppgave med de vanlige to poengene, eller legge mer arbeid i det med håp om å få med ekstrapoenget også.

Testfiler for komprimering

Oppgave (lyx)	https://www.idi.ntnu.no/emner/idatt2101/kompr/opg8-kompr.lyx
Forelesning (txt)	https://www.idi.ntnu.no/emner/idatt2101/kompr/diverse.txt
Forelesning (lyx)	https://www.idi.ntnu.no/emner/idatt2101/kompr/diverse.lyx
100MB enwik8	https://www.idi.ntnu.no/emner/idatt2101/kompr/enwik8

enwik8 er en diger fil, de første 100MB av engelsk wikipedia. Dere trenger ikke håndtere den for å få full uttelling, 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 minst to av testfilene.
2. Bruk en eller flere av algoritmene vi har sett på. Å kombinere flere, gir ofte bedre resultat. Dette kan gjøres med å lage flere uavhengige programmer, eller ett program som implementerer flere teknikker.
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. Test selv, før dere leverer øvingen.
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 kjøring. Det er *ikke* greit å ha ett samlepogram som både gjør innpakking og utpakking i *en* operasjon! Utpakking skal *bare* trenge den komprimerte fila, ikke noen variabler/datastrukturer fra innpakkinga. Det sikres ved at inn- og utpakking *ikke* skjer i samme kjøring.

Dere *kan* ha ett stort program som gjør hele jobben, men det må i så fall være i to separate kjøring. Et slikt program må altså stoppe etter komprimering, og kjøres igang med andre parametre for å pakke ut. Eller dere kan ha separate programmer.
7. Programmene må lese og skrive *filer*. Altså ikke bare testdata i en tabell. Filene ligger i den mappa programmet kjører i. Jeg må kunne teste på forskjellige filer.

8. Programmet bruker *ikke* Hasmap, Hashset eller andre former for hashtabeller. (Dette for å unngå de vanligste kokeoppskriftene, som bruker hashing.) Det er ikke bruk for hashing her – ingen hashing i eksemplene jeg har vist dere.

Tips for best kompresjon

Det enkleste er å gjøre Huffman- og LZ-delene helt separat. Man kan 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. Huffman-kompresjon fungerer vanligvis ikke så godt på kodene for repeterte strenger, så et smart Huffman-program kan skrive disse delene som de er, uten å forsøke å komprimere dem. Et slikt opplegg blir som regel bedre, men er noe mer arbeid å kode.

Når LZ ser etter repeterte strenger, vil kompresjon avhenge av hvor langt bakover LZ leter, og hvor kompakte referanser dere klarer å lage. Her er det rom for eksperimentering!

Run-length coding er enkelt, men oppnår ikke mye på testfilene.

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.

Om dere bruker flere teknikker, som LZ+Huffmann, er der lurt å teste og lage LZ-delen og Huffmann-delen hver for seg. Programmer som zip gjør alt i memory for å være raske. 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.

Tips om Lempel-Ziv

Filer

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 tekstsøkalgoritmer.

Filformat

Filformat bestemmer dere selv. Det kan fort bli en avveining 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...

Litt om å kombinere LZ og Huffman

Det enkleste er å gjøre de to delene separat. I så fall komprimerer man med LZ først, og deretter med Huffman. Prøv gjerne omvendt, men ikke regn med god kompresjon på det viset.

Man kan få bedre kompresjon ved å se på hva slags data LZ lager. Referanser bakover er et antall byte som blir temmelig tilfeldig. Tilfeldige data komprimeres ikke bra med Huffman, så det er bedre å holde slikt utenfor slik at Huffman bare komprimerer data og ikke disse referansene. Antall byte som refereres har derimot en kjent fordeling; det er mange fler referanser til korte data (ord o.l.) enn lange strenger. Så Huffmannkomprimering kan brukes på strenglengder. Beste måte er i så fall å ha ett huffmantre for data, og et annet tre for strenglengder. Da kan begge deler kodes med ganske korte koder.

Litt om Lempel-Ziv-Welsh

LZW bygger opp en ordbok/dictionary, og sparer plass ved å angi ordnumre i stedet for å lagre ordene i output. Til å begynne med inneholder ordboka de 256 mulige enkeltbytes, som har numre fra 0 til 255. Kompresjon oppnås ved å legge til ord (eller bokstavkombinasjoner) som forekommer ofte. Ulike måter for å legge til ord eller kortere sekvenser, gir ulik kompresjon. Man kan legge inn hver eneste sekvens som dukker opp: man får raskt noe å komprimere, men får fort så mange «ord» at det trengs mange bits for å referere til dem. En annen måte er å legge inn nye ord bare når de forekommer for andre eller tredje gang. Færre ord i ordboka, men de man får, forekommer oftere. Det kan gi bedre kompresjon.

Man lagrer aldri ordboka. Hele poenget med LZW er at utpakkingsdelen kan bygge opp ordboka på samme måte som innpakkingsdelen gjorde, ved å se på hva som har blitt pakket ut så langt. Plass spares ved at numrene som regel er kortere enn ord/tegnsekvenser de refererer til.

Kombinere LZW og Huffman

LZW lager en output som består av numre. I begynnelsen numre for diverse bytes, etterhvert også numre for ord som LZW har oppdaget. Man kan telle opp hvor mange det er av hvert nummer, og lage et Huffmantre. De ord/bytes som forekommer ofte, får da korte Huffmankoder og vi sparer mer plass.

Om Huffmankoding

Lag et program som leser inn en fil, teller frekvenser og genererer en huffmantre ut fra byteverdiene i filen. Deretter bruker programmet huffmantreet til å skrive en komprimert huffmannkodet 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 løsning. For å slippe overhead med en lagret tabell, skriver man heller en «blokk» uten kompresjon. Neste blokk komprimeres med et huffmantre basert på forrige blokk. Poenget med dette, er at utpakkingsprogrammet kan gjøre det samme. Et problem med slike løsninger, er at en blokk kan inneholde tegn som ikke var i den forrige blokka, og det må håndteres. En måte er å gi tegn som ikke er med, en lav frekvens i stedet for 0. En annen måte er å ha en escape-kode for slike tilfeller. Når programmet dekode en escapekode, leser det deretter inn et tegn som ikke er komprimert med huffmankoding.

Tips om Huffmankoding

Huffmanndata som trengs for å pakke ut igjen

Det er ikke nødvendig å lagre huffmantreet, 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. Testfilene 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.

Datatypes «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 huffmann-treet.

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. Hashtabeller trengs ikke for å løse denne oppgaven.

Delvis uttelling

Ikke gi opp, om hele oppgaven blir for vanskelig. Delvis uttelling for oppgaver som:

- Har noen få feil ved utpakking. Her må jeg kunne se at fila som pakkes ut har mye av det samme innholdet, så det går ikke med bare rot. Men noen feil her og der, holder til delvis uttelling.
- Klarer å pakke inn/ut deres egne testfiler, men ikke mine. Send i så fall med en testfil som fungerer. Det hender f.eks. at noen lager programmer som bare håndterer ascii-tekst.

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, bitoperasjoner og filbehandling

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