Institut for Matematik og Datalogi
Syddansk Universitet, Odense

23. oktober 2001
KSL

# DM02 eksamen – Obligatorisk Opgave

Kim Skak Larsen
Efteråret 2001

## Indledning

I denne note beskrives den obligatoriske opgave, der skal løses i forbindelse med DM02, efteråret 2001. Efter en indledende beskrivelse af det problem, der skal arbejdes med, beskrives input/output-formater mm. Læg mærke til, at hele opgaven skal afleveres på papir i form af en rapport, men at programmet også skal afleveres elektronisk. Programmerne vil blive afprøvet automatisk, og det er derfor utroligt vigtigt at overholde kravene til input/output-format med videre. I teksten beskrives det, hvordan man selv kan kontrollere, at ens format er i orden. Til sidst beskrives afleveringsproceduren. Det er vigtigt at læse hele opgaven igennem, før man begynder sit arbejde.

## Filkompression

Når man har et lagermedie, som f.eks. en harddisk eller en CD-rom, vil man gerne udnytte sin plads så godt som muligt. Når man henter information via Internettet, er man pga. den lave transmissionshastighed interesseret i, at den information, man henter hjem, fylder så lidt som muligt.

Man er altså generelt interesseret i at kunne komprimere information (før det gemmes eller sendes). I princippet er det underordnet, om den komprimerede information kan læses (af en person), når blot man senere kan ekspandere det komprimerede format tilbage til det oprindelige.

Opgaven drejer sig om denne problematik, som vi vil referere til som filkompression. Der skal laves et program, der kan læse en fil og konstruere en komprimeret udgave af filen. Programmet skal naturligvis også kunne rekonstruere filen igen ud fra den komprimerede version. Disse funktionaliteter kunne kaldes *compress* og *uncompress*.

Faktisk skal der laves to udgaver af ovenstående: en debug-version samt en rigtig version. Det giver anledning til fire funktionaliteter, som vi i det følgende vil referere til som *debug-compress*, *debug-uncompress*, *binary-compress* og *binary-uncompress*.

## Mulighed for forbedring?

De fleste dataformater til at udtrykke tegnsæt i baserer sig på et fast antal bits. Det mest kendte er ASCII-systemet, der bruger 7 bits. Dermed kan man indkode $2^7 = 128$ tegn. Ved at skrive `man ascii` i en shell kan man se, hvad disse 128 tegn er. Som et andet eksempel baserer Java sig internt på 16 bits til sin `char` type.

Systemer baserer sig oftest på et fast antal bytes, der er betegnelsen for 8 bits. Det er dog ikke altid, at alle 8 bit bliver udnyttet (som i ASCII systemet). I denne opgave lægger vi os fast på at betragte de filer, vi komprimerer, som bestående af et antal bytes; dvs. grupper af 8 bits.

Se nu på en tekst som i Figur 1.

```
to be or not to be
```

Figur 1: Eksempel på input.

Ved at checke i en ASCII tabel kan man se, at "t" er tegn nummer 116 (01110100 binært med brug af 8 bit), "o" er tegn nummer 111 (det tal er fem mindre, da tegnene er nummeret i alfabetisk orden), osv. Ud over de sædvanlige alfabetiske tegn vil ovenstående eksempel lagret på en fil indeholde fem blanke (mellem de seks ord). En blank er tegn nummer 32 (00100000). I en almindelig tekstfil afsluttes alle linier af et "newline" tegn, som er tegn nummer 10 (00001010). Et "newline" tegn noteres som regel som "\n".

Da vores Linux system, som de fleste andre, baserer sig på ASCII tegnene, vil vi forvente, at eksemplet i virkeligheden opbevares som den sekvens af bits, man kan se i Figur 2:

| | | | | |
|---|---|---|---|---|
| 01110100 | 01101111 | 00100000 | 01100010 | 01100101 |
| 00100000 | 01101111 | 01110010 | 00100000 | 01101110 |
| 01101111 | 01110100 | 00100000 | 01110100 | 01101111 |
| 00100000 | 01100010 | 01100101 | 00001010 | |

Figur 2: Eksemplet i standardrepræsentation.

Teksten fylder 19 bytes (da der er 19 tegn).

## Huffman-træer

Komprimering vha. Huffman-træer er et opgør med den tanke, at ethvert tegn skal repræsenteres af det samme antal bits. Ideén er at lade ofte forekommende tegn få en kort bit-kode.

Vi vil først diskutere konstruktionen af Huffman-træer med fokus på vores anvendelse af dem.

Der tages udgangspunkt i en sekvens af tegn (dem som forekommer i den tekst, der skal komprimeres) med en angivelse af, hvor ofte de hver især forekommet.

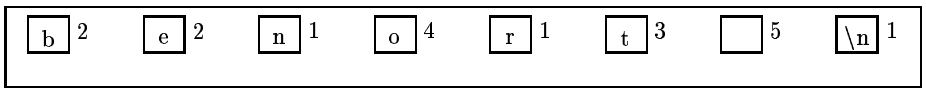
Tabellen hørende til ovenstående eksempel ses i Figur 3.

| b | e | n | o | r | t | ⟨space⟩ | ⟨newline⟩ |
|---|---|---|---|---|---|---|---|
| 2 | 2 | 1 | 4 | 1 | 3 | 5 | 1 |

Figur 3: Forekomsttabel for eksemplet.

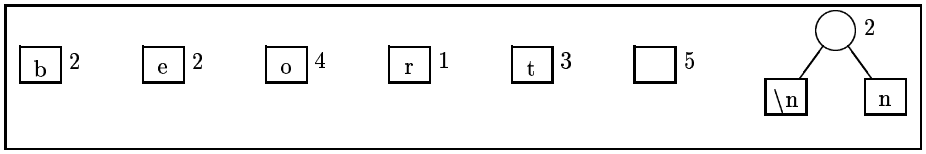
Konstruktionen af Huffman-træet er en iterativ process, hvor man sammensætter træer med vægte. Man tager hele tiden to træer med mindst vægt og sætter dem sammen ved at gøre dem til børn af en ny rod. Det nye træ får vægt lig med summen af de to træers vægte. Der fortsættes til kun ét træ er tilbage.

De træer, der tages udgangspunkt i, er ét træ for hvert tegn. Sådan et træ består af kun én knude. Træet har vægt lig med tegnets forekomsttal fra tabellen. Udgangspunktet ses i Figure 4.




Figur 4: Udgangspunktet for konstruktionsalgoritmen.

Vi skal nu tage to træer med mindst vægt og sætte dem sammen. Der er tre træer med vægt én. Vi vælger (vilkårligt), at sætte "n" og "\n" sammen. Det giver den nye samling træer, der ses i Figur 5.

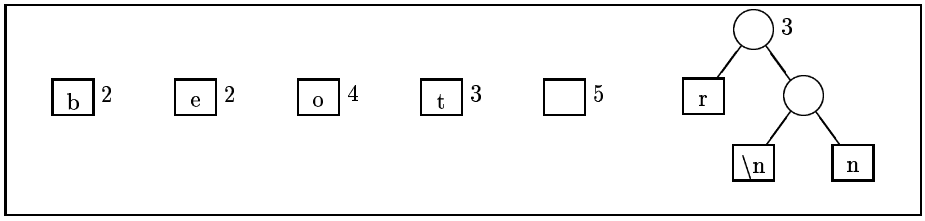



Figur 5: Efter 1. iteration.

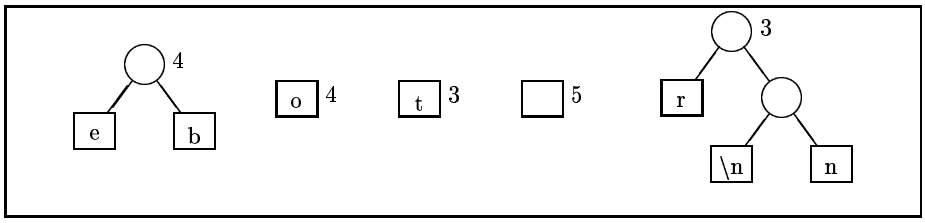
Nu er der ét træ med vægt én, som vi nødvendigvis skal tage, og så skal vi vælge et af træerne med vægt to at sætte sammen med det. Vi vælger (vilkårligt) at sætte "r" træet sammen med det træ og får situationen i Figur 6.

Sådan fortsættes og vi får den sekvens af træsamlinger, der ses i Figurerne 7, 8, 9, 10 og 11.

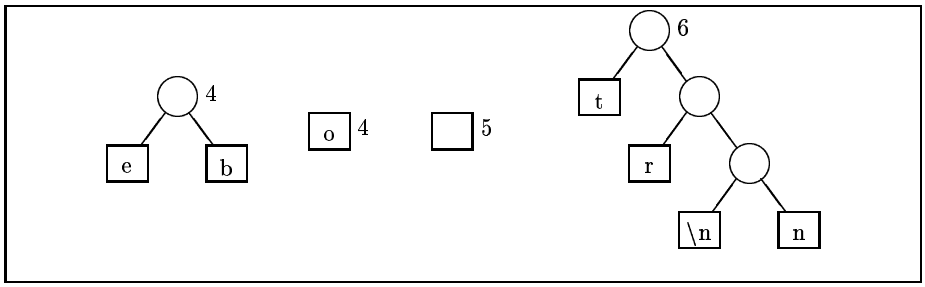
Efter konstruktionen er vægten af træet ikke længere interessant. Det endelige træ ses i Figur 12.



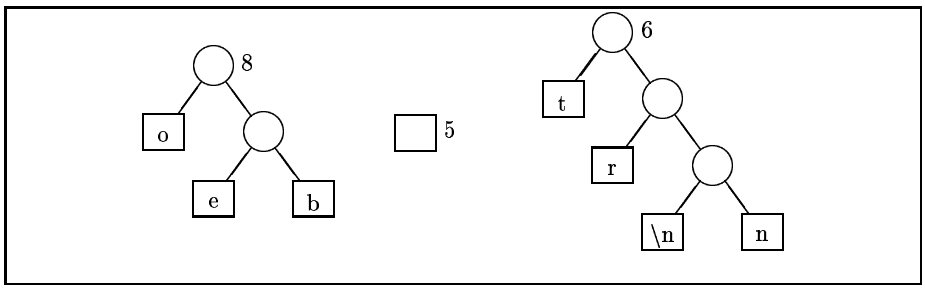

Figur 6: Efter 2. iteration.



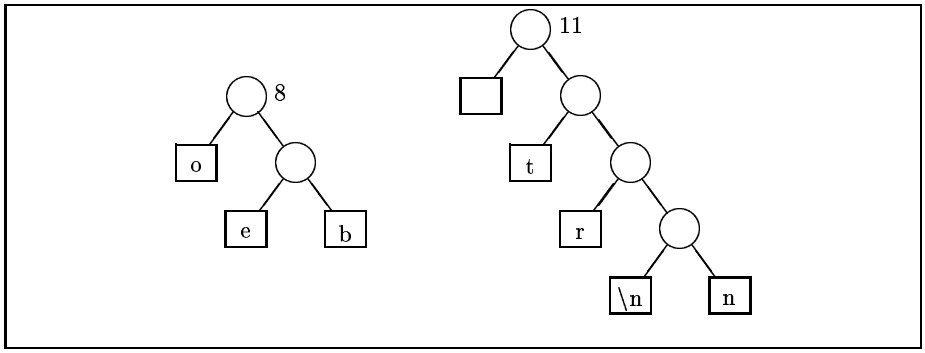

Figur 7: Efter 3. iteration.



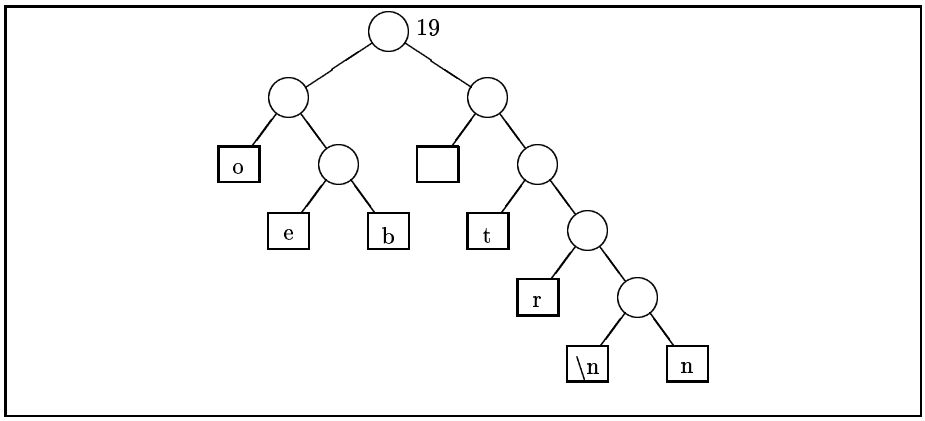

Figur 8: Efter 4. iteration.



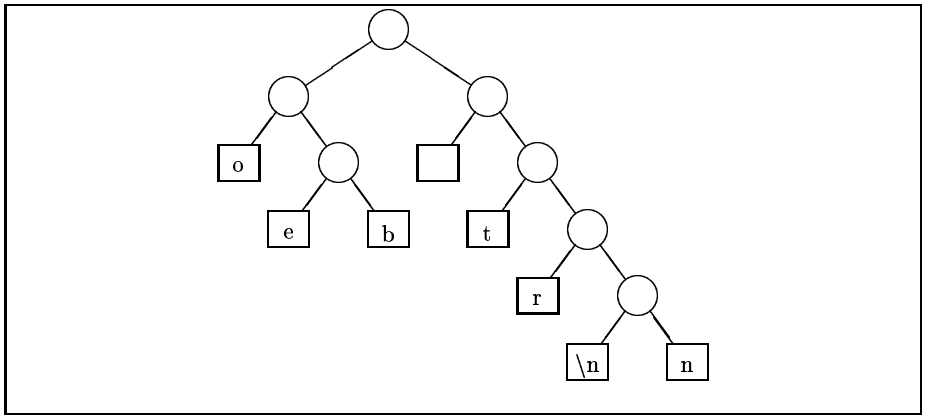

Figur 9: Efter 5. iteration.




Figur 10: Efter 6. iteration.




Figur 11: Efter 7. iteration.




Figur 12: Det endelige træ.

Det er vigtigt at bemærke, at der kan være flere korrekte Huffman-træer, fordi man blandt flere træer med samme vægt kan vælge vilkårligt. Hvis man vælger anderledes, får man et andet resultat. Det er dog sådan, at længden af komprimeringen bliver den samme.

Indkodningen af de enkelte tegn kan nu læses ud af træet i Figur 11, idet vi bestemmer os for at associere bit'en "0" med en venstregren og bit'en "1" med en højregren.

For eksempel indkodes "e" som `010`, en blank indkodes som `10` og "n" indkodes som `11111`.

Den indkodede tekst ses i Figur 13.

```
110 00 10 011 010 10 00 1110 10 11111 00 110 10 110 00 10 011 010 11110
```

Figur 13: Den indkodede tekst.

Dvs. 53 bits som kan pakkes i $\lceil 53/8 \rceil = 7$ bytes.

Der er selvfølgelig det lille problem, at dette ikke er en standard indkodning, men én vi selv har fundet på, så nok kan vi gemme det komprimeret i 7 bytes, men vi kan ikke læse det igen!

For at kunne dekode informationen skal vi bruge træet, så det skal altså også gemmes. Det betyder, at på dette lille eksempel er der ikke nogen gevinst, men på store datamængder udgør træet en relativ lille procentdel og så kan der spares megen plads. Træets størrelse er jo højst proportionalt med antallet af tegn, man har lov til at bruge (som er en konstant).

Den komprimerede fil består altså af en repræsentation af Huffman-træet fulgt af en indkodning af originalteksten konstrueret i følge Huffman-træet.

## Input, output og kørsler

I dette afsnit beskrives formatet for input og output samt præcis, hvordan programmerne skal kunne afvikles.

### Input

Som input til debug-compress eller binary-compress skal kunne tages enhver fil, der indeholder mindst to forskellige tegn, og som ikke indeholder nogen forekomster af tegnene med ASCII-værdierne 29 og 30.

Kravet om to forskellige tegn sikrer, at I ikke skal behandle et uinteressant lille specialtilfælde.

Kravet omkring ASCII-værdierne 29 og 30 er teknisk (se næste afsnit) og gør også nogle ting nemmere at programmere. Almindelige tekstfiler indeholder ikke de to tegn.

Input til debug-uncompress og binary-uncompress skal være et tidligere output fra henholdsvis debug-compress og binary-compress.

### Output

Først beskrives output formatet for debug-compress. Dette output kan betragtes som et mellemresultat før den endelige komprimering.

Først laves en repræsentation af det Huffman-træ, man har opbygget. Dette gøres vha. et post-order gennemløb af Huffman-træet, hvor der udskrives forskellig information for knuderne afhængig af, om de er blade eller interne knuder. Informationen for en given knude udskrives på en linie for sig. For blade, der jo indeholder et tegn, udskrives tegnets ASCII-værdi (decimalt). For interne knuder udskrives tegnet "@" på en linie for sig.

Efter trærepræsentationen skrives "@@" på en linie for sig. Nu følger den indkodede tekst, der som bekendt er en følge af bits. Disse udskrives med 8 bit på hver linie. Der er naturligvis ingen garanti for, at 8 går op i længde af den indkodede tekst. Hvis den ikke gør, fyldes der på med nuller, så også den sidste gruppe bits har en størrelse på 8. Som det allersidste skrives et tal mellem 1 og 8 (begge inklusive), der angiver, hvor mange af bit'ene i den sidste gruppe, man kan stole på (tallet er altså 8 minus antallet af nuller, man hæftede på). I Figur 14 ses et muligt output fra debug-compress kørt på eksemplet fra Figur 1.

Det træ, der er udskrevet ved et post-order gennemløb, er altså det, der ses i Figur 12. Tallet 5 i sidste linie angiver, at kun bit'ene 11110 fra den andensidste linie (11110000) skal bruges i dekodningen (sammenlign med bit'ene i Figur 13).

Nu beskrives output-formatet for binary-compress. Dette output-format er næsten som det for debug-compress, men nu skal der naturligvis virkelig komprimeres, så alt skal gøres så kompakt som muligt.

Først droppes linieskift ("newline"). Dvs. alt skrives på én lang linie afsluttet af ét linieskift til allersidst.

Bladene i træet skrives som bytes; én byte pr. blad. Dvs. at for eksempel linien "114" af en byte bestående af det bitmønster (01110010), der svarer til tallet 114. I stedet for "@" skrives en byte svarende til det decimale tal 29 og i stedet for "@@" skrives en byte svarende til det decimale tal 30. Disse tal (bitmønstre) er valgt, fordi de ikke bliver anvendt som koder for tegn i almindelige tekstfiler. Det er naturligvis vigtigt, for ellers kan vi i dekodningen ikke afgøre, om vi har fat i en intern knude eller et blad.

Grupperne af 8 bit skrives naturligvis som én byte hver. Det sidste tegn ("5" i eksemplet) skrives binært (dvs. som en byte svarende til mønstret 00000101).

Det er lidt vanskeligere at illustrere et binært output, da det ser meget underligt ud på skærmen. Problemet er, at de programmer, vi normalt bruger til at vise tekst med, jo ikke kan vide, at vi har lavet vores egen kodning, så de programmer vil blot vise deres fortolkning af de bytes, de ser. Angivet som en sekvens af bytes vist som decimaltal indeholder den komprimerede fil det, der vises i Figur 15.

```
111
101
98
@
@
32
116
114
10
110
@
@
@
@
@
@@
11000100
11010100
01110101
11110011
01011000
10011010
11110000
5
```

Figur 14: Eksempel på output fra debug-compress.

```
111 101 98 29 29 32 116 114 10 110 29 29 29 29 29 30 196 212 117 243 88
154 240 5 10
```

Figur 15: Illustration af output fra binary-compress.

**Kørsler**

Der er følgende krav til afviklingen af det færdige Java-program. Ens hovedfil skal hedde `Huffman.java` (stort 'H'). Man skal kunne afvikle programmet på følgende fire måder:

```
java Huffman cd infilename outfilename
java Huffman ud infilename outfilename
java Huffman c infilename outfilename
java Huffman u infilename outfilename
```

svarende til de fire funktionaliteter: `c` står for compress, `u` for uncompress og `d` for debug. Input'et findes på `infilename`, og output skal skrives på `outfilename`. I kataloget `/home/IMADA/courses/dm02/Tests` findes input-filer, man kan øve sig på. Disse filnavne ender alle på `.test`.

Hvis I gerne vil se, hvad vi senere vil sige til det output, I producerer, kan I skrive `DM02check`. I skal, før I afgiver kommandoen, placere jer i det katalog, alle jeres oversatte Java-program ligger i. Så vil jeres program blive kørt på samtlige input-filer i `/home/IMADA/courses/dm02/Tests`, og resultaterne bliver kommenteret. Hvis der er fejl, vil kun første fejl for hver testfil blive rapporteret. Hvis der ikke findes fejl, betyder det ikke nødvendigvis, at jeres program fungerer helt korrekt, da vi ikke kan kontrollere alt automatisk. Programmet `DM02check` kan afbrydes med `Ctrl-C`.

For alle filer med endelsen `.test` i `/home/IMADA/courses/dm02/Tests` laver `DM02check` en række filer i jeres eget katalog indeholdende output fra jeres program kørt på testfilerne. Disse filer ender alle på `.out`. Bemærk, at en sådan fil kun laves, hvis I ikke allerede har en sådan fil (af sikkerhedsgrunde; vi vil jo ikke overskrive jeres egne filer). Hvis I foretager rettelser og gerne vil køre `DM02check` igen, skal I altså først fjerne alle de gamle `.out`-filer.

Hvis det tager for lang tid at afvikle en enkelt test, vil det af `DM02check` blive betragtet som en fejl (en uendelig løkke), og det vil føre til en afbrydelse.

## Specifikke krav til rapport og program

Implementationen skal foretages i overensstemmelse med beskrivelserne givet i denne opgavetekst. Under konstruktionen af et Huffman-træ skal der anvendes en prioritetskø til opbevaring af deltræerne (træernes vægte er naturligvis prioriteterne).

Der vil blive lagt vægt på, at der anvendes effektive algoritmer (som gennemgået i DM01 og DM02) til løsning af delproblemer, som realisering af en prioritetskø, post-order gennemløb af et træ, osv.

Rapporten skal indeholde en beskrivelse af de væsenligste valg, der er truffet i forbindelse med implementationen, samt begrundelser herfor. Derudover skal det

undersøges, hvor godt denne kompressionsmetode virker (dette drejer sig udelukkende om funktionaliteten binary-compress). Kompression måles i procent på følgende måde. Antag, at den oprindelige fil fylder $b$ bytes og den komprimerede $c$ bytes. Så er filen blevet komprimeret $\frac{b-c}{b} \cdot 100\%$. Det skal undersøges på hvilke typer filer, metoden virker bedst. Prøv mindst tre forskellige typer, f.eks. Java-filer, html-filer, almindelig dansk tekst, almindelig engelsk tekst, osv. For hver type skal effekten undersøges for filer af forskellig størrelse, så man kan konkludere, hvordan metoden fungerer på virkelig store filer.

Man skal også angive tidskompleksiteten af ens løsning; gerne med henvisning til undervisningsmaterialet i det omfang, det er muligt. Endelig skal rapporten indeholde en udskrift af hele programmet.

Programmet skal skrives i JAVA og skal køre på IMADA's maskiner.

Der er følgende krav til JAVA-programmerne:

- Alle filer skal ligge i ét katalog (directory) hos jer selv på IMADA's system.
- Hovedprogrammet skal hedde `Huffman.java` (stort 'H').
- Kørsler skal kunne foretages som beskrevet i foregående afsnit (igen på IMADA's system).

# Hjælp

I dette afsnit gives der hjælp af forskellig slags. Dels foreslås en programstruktur, som det kraftigt anbefales, man anvender. Dels beskrives en række JAVA-konstruktioner og andre ting, der *muligvis* kan vise sig nyttige, og som jeg ikke ønsker, I skal spilde tid med selv at finde ud af. Det er ikke sikkert, at I alle får brug for disse ting. Det kommer ganske an på, hvordan I løser opgaven.

## Forslag til programstruktur

Det foreslås, at man først laver debug-delen og får funktionaliteterne debug-compress og debug-uncompress i orden. Lad være med at skrive direkte ud til filen. Opsaml i stedet output som en sekvens (array eller Vector) af strenge og skriv så ud til allersidst.

Hvis man følger ovenstående kan man lettere tilføje den rigtige (binære) kompression. Der laves simpelt hen to funktioner `debug2binary` og `binary2debug`, der konverterer mellem de to formater. Funktionen `debug2binary` tager en sekvens af strenge repræsenterende et debug-output og konstruerer det tilsvarende binære output ud. Funktionen `binary2debug` tager som input én lang sekvens repræsenterende en binær indkodning og returnerer en sekvens af strenge, der udgør den ækvivalente debug-indkodning.

Fordelen ved ovenstående er, at I kan få langt størstedelen af programmet til at virke uden at skulle bekymre jer om den binære del.

## Java-konstruktioner

Indlæsning af filer kan klares med

```
FileReader fr = new FileReader(fileName);
BufferedReader inFile = new BufferedReader(fr);
```

hvorefter man kan læse en byte (8 bit) af gangen med

```
i = inFile.read();
```

Disse bit bliver dog returneret i form af en `integer` (derfor variabelnavnet `i`).

Skrivning af filer klares tilsvarende med

```
FileWriter fw = new FileWriter(fileName);
BufferedWriter outFile = new BufferedWriter(fw);
```

og f.eks.

```
outFile.write(i);
```

for at skrive en byte og

```
outFile.write(line, 0, line.length());
```

for at skrive en linie.

Husk at lukke alle filer efter endt brug. F.eks. lukkes `outFile` ved at udføre `outFile.close()`.

## Andet

Der findes manual-sider, der kan bruges ved at skrive `man kommando`, hvor `kommando` er den kommando, man gerne vil vide noget om.

For at se hvor meget en fil fylder, kan man i Linux skrive `ls -l filename`. Tallet før datoen angiver antallet af bytes. En anden mulighed er `wc -c filename`.

For at se hvad der er på en binær fil, kan man bruge f.eks. `od -Ad -tc -tuC filename`.

Får man brug for at konvertere et decimalt tal til en bitstreng, kan følgende metode (her illustreret ved et eksempel) benyttes. Vi prøver med tallet 13. Nu laver vi heltalsdivision med 2, indtil vi når ned på nul: $\frac{13}{2} = 6$, $\frac{6}{2} = 3$, $\frac{3}{2} = 1$, $\frac{1}{2} = 0$. Hvis vi ser på denne sekvens af udregninger *bagfra* og skriver 0, når divisionen går op, og 1 når den ikke gør, fås 1101, som er den binære repræsentation af det decimale tale 13. Dette er ikke et tilfælde. `:-)`

## Generelle krav til design, rapport og program

Først og fremmest skal alle krav beskrevet i denne opgaveformulering naturligvis tilfredsstilles.

Programmet skal være velstruktureret og kommenteret i passende omfang.

Rapporten skal indeholde en udskrift af hele programmet. Denne udskrift skal være identisk med det elektronisk afleverede program.

Rapporten skal indeholde en beskrivelse af de væsentligste valg, der er truffet i forbindelse med implementationen, samt begrundelser herfor.

Desuden skal det forklares, hvordan programmet er afprøvet. Testeksempler og testkørsler kan vedlægges i det omfang, det er meningsfyldt (rigtigt store filer vedlægges ikke i udskrift).

Eventuelle mangler i program eller rapport skal beskrives.

Endelig skal rapporten underskrives.

## Aflevering

Der skal afleveres en rapport på sekretariatet, og et program skal afleveres elektronisk.

Den elektroniske aflevering foregår på følgende måde: I afleverer ved først at placere jer i det katalog (directory), der indeholder alle jeres Java-filer til opgaven, hvorefter I afgiver kommandoen `DM02aflever`. Denne aflevering skal foretages inden mandag d. 12. november 2001 kl. 12:00. Bemærk, at I (inden denne frist) kan aflevere flere gange, hvis I fortryder en aflevering. Kun den sidste aflevering tæller (de andre slettes).

Rapporten afleveres på sekretariatet på Institut for Matematik og Datalogi senest mandag d. 19. november 2001 kl. 12:00. Denne opgaveformulering er vedhæftet en forside, som skal anvendes ved afleveringen. Husk for jeres egen skyld at få en kvittering (også vedhæftet) på, at I har afleveret.

Husk, at det er et krav, at det elektronisk afleverede program er præcis det samme program, der afleveres en udskrift af i rapporten.

## Yderligere formalia

Den obligatoriske opgave er et individuelt eksamensprojekt. Der må altså ikke arbejdes i grupper. En overtrædelse af dette er eksamenssnyd og vil blive behandlet som sådan. Man har pligt til selv at beskytte sine noter og filer mod læsning af andre. Dette kan f.eks. gøres med `chmod 700 opgavedirectory`. Begge parter involveret i en eventuel plagiering kan blive holdt ansvarlige.

Følgende gælder både for rapportaflevering, samt en evt. elektronisk aflevering.

Der gives altid betragteligt mere tid til opgaverne end nødvendigt. Til gengæld accepteres for sent afleverede opgaver ikke; heller ikke bare 5 minutter!

Det anbefales at man færdiggør og afleverer opgaven mindst 48 timer før afleveringsfristen. At systemet eller alle printere eksempelvis er nede de sidste 24 timer før afleveringsfristen betragtes ikke som en legal undskyldning. Det samme gælder syge børn, egen sygdom uden lægeerklæring, osv.

Eventuelle programmer skal kunne køres på IMADA's maskiner. Man må gerne lave dem hjemme, men det er helt på eget ansvar. Tekniske problemer derhjemme er ingen undskyldning. Man er også helt selv ansvarlig for, at ens filer kan flyttes uden problemer. At f.eks. modem-forbindelsen er nede, diskettedrevene ikke virker, filformaterne eller e-mail protokollerne ikke stemmer overens er ens eget problem.

Institut for Matematik og Datalogi
Odense Universitet

23. oktober 2001
KSL

# DM02 eksamen, Efteråret 2001
# Obligatorisk Opgave

**Skriv tydeligt** (maskinskrift eller blokbogstaver)

Navn: .....

Fødselsdato: .....

Brugernavn (login): .....

| **Instruktor** | Ditte | Henning | Jens |
|---|---|---|---|
| **Sæt kryds** | | | |

Besvarelsen omfatter ..... nummererede sider.

# Kvittering for aflevering af obligatorisk opgave i DM02, efteråret 2001

**Udfyldes inden afleveringen**

Navn: .....

Fødselsdato: .....

Brugernavn (login): .....

**Udfyldes af sekretariatet**

Modtaget den ..... kl. ..... af .....
(dato) (klokken) (initialer)