Institut for Matematik og Datalogi
Syddansk Universitet, Odense

4. oktober 2007
LMF

DM507 Eksamen — Obligatorisk Opgave
Et Pakningsproblem

1 Indledning

I denne note beskrives hele den obligatoriske opgave (1. + 2. del), der skal løses i forbindelse med DM507, efteråret 2007. Hele opgaven skal afleveres samlet senest

mandag d. 26. november kl 12:00.

Første delopgave gik ud på at implementere de fire bin packing algoritmer First-Fit, First-Fit-Decreasing, Next-Fit og Next-Fit-Decreasing. Anden delopgave går ud på at implementere algoritmen KNAPSACK, som baserer sig på en algoritme, der løser knapsack-problemet for hver enkelt kasse (bin).

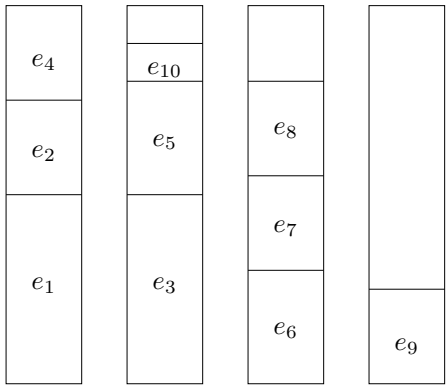
Først beskrives bin packing problemet (afsnit 2). Derefter beskrives de fire algoritmer fra første delopgave samt algoritmen KNAPSACK (afsnit 3). Afsnit 4 beskriver input-/output-formater mm. Læg mærke til, at *output-formatet for* KNAPSACK *er lidt anderledes end for de fire andre algoritmer*. Programmerne vil blive afprøvet automatisk, og det er derfor vigtigt at overholde kravene til input/output-format med videre. I afsnit 5 beskrives det, hvordan du selv kan kontrollere, at dit format er i orden. I afsnit 6 bliver du bedt om at overveje, hvor godt de enkelte algoritmer løser problemet. Kravene til rapporten er beskrevet i afsnit 7, og afsnit 8 gengiver afleveringsproceduren. Endelig understreges det i afsnit 9, at projektet skal løses individuelt. Det er vigtigt at læse hele opgaven igennem, før man begynder sit arbejde.

2 Paknings-problemet

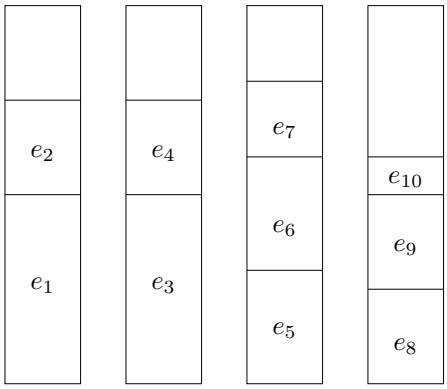
I denne opgave ser vi på et pakningsproblem, hvor vi er givet en mængde af kasser og en mængde af elementer, som skal pakkes i kasserne. Kasserne har alle størrelse B, men elementerne kan have varierende størrelse. Det gælder om at bruge *så få kasser som muligt* uden at overfylde nogen kasse; dvs. den samlede størrelse af elementerne pakket i en enkelt kasse må ikke overstige B.

3 Algoritmerne

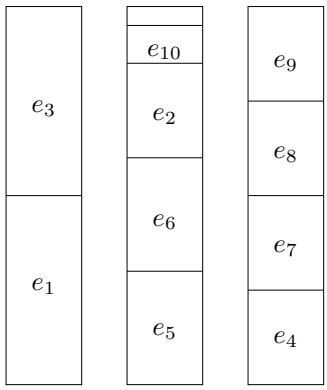
Det beskrevne pakningsproblem er NP-fuldstændigt. Det betyder, at man regner med, at der ikke findes nogen polynomiel algoritme, som løser problemet optimalt. (At en algoritme er polynomiel betyder, at der findes en konstant c, så algoritmen har køretid $O(n^c)$.) Derfor må vi ty til algoritmer, som ikke altid finder den bedst mulige løsning. Algoritmerne, som vi skal se på i denne opgave, er First-Fit (FF), First-Fit-Decreasing (FFD), Next-Fit (NF), Next-Fit-Decreasing (NFD) og KNAPSACK. De første fire er polynomielle algoritmer, hvorimod den sidste har det, man kalder pseudo-polynomiel køretid.



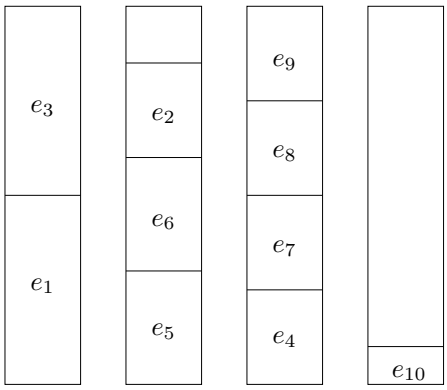
(a) First-Fit



(b) Next-Fit



(c) First-Fit-Decreasing



(d) Next-Fit-Decreasing

Figur 1: De fire polynomielle algoritmer anvendt på sekvensen $\langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10} \rangle$, hvor $|e_1| = |e_3| = 50$, $|e_5| = |e_6| = 30$, $|e_2| = |e_4| = |e_7| = |e_8| = |e_9| = 25$, $|e_{10}| = 10$ og $B = 100$.

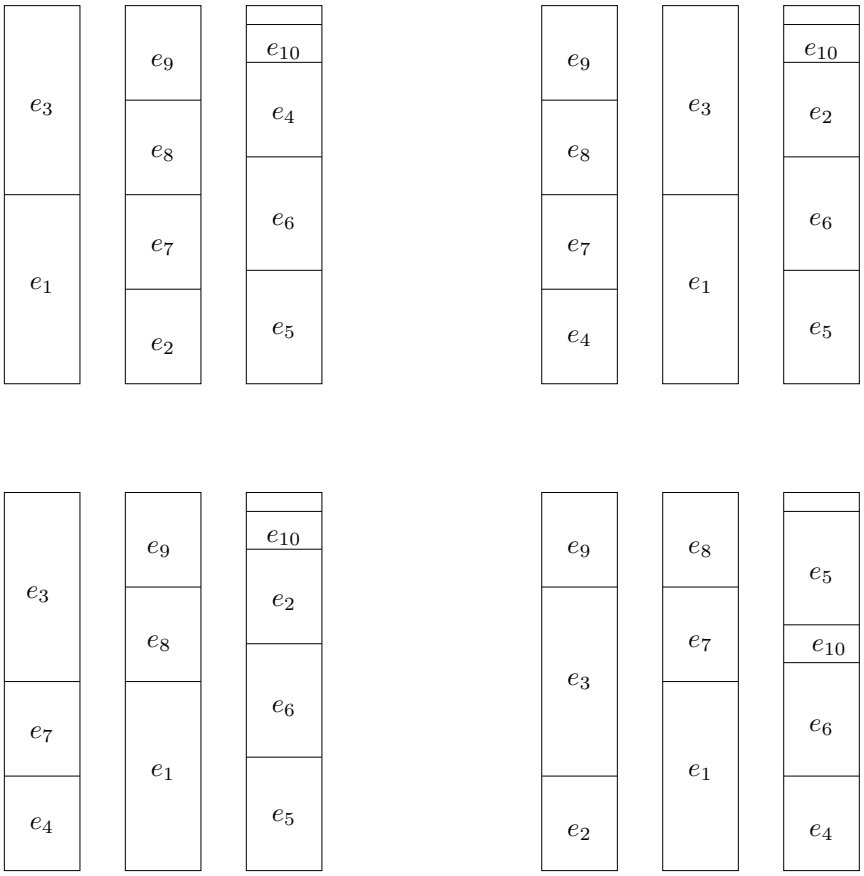
3.1 Polynomielle algoritmer

First-Fit nummererer kasserne efter den rækkefølge, den tager dem i brug. Den ser på elementerne et for et, og pakker hvert element i den første kasse, som har plads nok. Er der ingen af de allerede anvendte kasser, som har plads til elementet, tages en ny kasse i brug.

Next-Fit ser også på elementerne et efter et. Hvis det aktuelle element passer i den kasse, som er taget sidst i brug, pakkes det der. Ellers tages en ny kasse i brug.

First-Fit-Decreasing gør det samme som First-Fit, bortset fra at den starter med at sortere elementerne efter størrelse, og pakker de største elementer først. Tilsvarende gør *Next-Fit-Decreasing* det samme som Next-Fit, bortset fra at den pakker elementerne i rækkefølge efter faldende størrelse.

I figur 1 er vist et eksempel, hvor kasserne har størrelse 100, og elementerne har størrelse 50, 25, 50, 25, 30, 30, 25, 25, 25, 10.



Figur 2: Fire lovlige KNAPSACK-løsninger for input-sekvensen $\langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\rangle$, hvor $|e_1| = |e_3| = 50$, $|e_5| = |e_6| = 30$, $|e_2| = |e_4| = |e_7| = |e_8| = |e_9| = 25$, $|e_{10}| = 10$ og $B = 100$.

3.2 Algoritmen KNAPSACK

Algoritmen KNAPSACK fylder hver kasse mest muligt, givet de tilbageværende elementer. Dvs. for hver kasse løses knapsack-problemet for de endnu ikke pakkede elementer. Kapaciteten af knapsack'en svarer til kassens størrelse B. Størrelsen af et element anvendes både som vægt (w) og værdi (v) af elementet.

For en given instans af bin packing problemet kan der være flere KNAPSACK-løsninger. Ser vi igen på eksemplet fra figur 1, ville samtlige løsninger illustreret i figur 2 f.eks. kunne produceres af en algoritme, som svarer til ovenstående beskrivelse. Dette skyldes, at der kan være flere delmængder af de tilbageværende elementer, som giver den optimale fyldningsgrad af en kasse.

Bemærk, at selvom algoritmen KNAPSACK fylder hver kasse mest muligt (givet de elementer, der er tilbage), bruger den ikke nødvendigvis det mindst mulige antal bins. Det ses bl.a. af følgende eksempel: Kasserne har størrelse $B = 100$. Der er 3 elementer af størrelse 30 og 3 elementer af størrelse 55. De 6 elementer kan pakkes i 3 kasser, men KNAPSACK bruger 4.

4 Programmet

Programmet skal skrives i JAVA og skal køre på IMADA's maskiner. Programmet skal være velstruktureret og kommenteret i passende omfang.

Algoritmerne skal implementeres sådan, at NF har køretid $O(n)$, NFD har køretid $O(n \log n)$, FF og FFD har køretid $O(n^2)$, og KNAPSACK har køretid $O(n^2 B)$, hvor n er antallet af elementer. Algoritmerne FF og FFD kan implementeres, så de har køretid $O(n \log n)$, men det er ikke et krav i denne opgave.

Skriv pseudokode for algoritmerne, inden du begynder at implementere dem.

Til at sortere elementerne må du gerne bruge JAVAs sorteringsmetode, men husk, at den sorterer elementerne i stigende orden, ikke i faldende.

4.1 Input, output og kørsler

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

Input

Første linie i input-filen er et heltal, som angiver størrelsen B af kasserne. På de næste n linier står størrelserne af de n elementer, der skal pakkes. Disse størrelser er heltal mellem 1 og B. Som en konkret illustration kan man se på input til eksemplet fra Figur 1.

```
100
50
25
50
25
30
30
25
25
25
10
```

Figur 3: Input svarende til eksemplet i Figur 1.

Du må gerne gå ud fra, at input overholder det beskrevne format. Dvs. *dit program behøver ikke at kunne håndtere forkert input.*

Output

Output'et består af fem dele svarende til de fire algoritmer FF, NF, FFD, NFD og KNAPSACK, i denne rækkefølge. Output'et fra de enkelte algoritmer adskilles med præcis en blank linie. Dvs. output'et indeholder præcis 4 blanke linier i alt.

For hver af de fire første algoritmer skal der være en linie for hver kasse, som tages i brug. På linie i står størrelserne af de elementer som pakkes i den i'te kasse. Elementerne skal optræde i den

rækkefølge, de blev pakket i. Der skal være præcis et mellemrum mellem hvert par af nabotal. Der må ikke være mellemrum efter det sidste tal på en linie. Hver linie (også den sidste) afsluttes med "\n".

For algoritmen KNAPSACK svarer linie i også til den i'te kasse, men i stedet for at skrive størrelsen af hvert element pakket i i'te kasse skrives blot den samlede størrelse af de elementer, som pakkes i denne kasse.

I Figur 4 kan man se det korrekte output for eksemplet fra Figur 1.

```
50 25 25
50 30 10
30 25 25
25

50 25
50 25
30 30 25
25 25 10

50 50
30 30 25 10
25 25 25 25

50 50
30 30 25
25 25 25 25
10

100
100
95
```

Figur 4: Output svarende til eksemplet i Figur 1.

Kørsler

Ud over input/output-format er der følgende krav til programmet:

- Alle filer skal ligge i ét katalog (directory) hos jer selv på IMADA's system.
- Hovedprogrammet skal hedde `BP.java` (for Bin Packing). Dvs. man skal kunne afvikle programmet på følgende måde:

  ```
  java BP <filename>
  ```

  hvor filen `filename` indeholder input.
- Programmet skal fungere på IMADAs system.

5 Test

Testen skal designes, inden du skriver programmet. Overvej, hvilke specialtilfælde man kan komme ud for, og hvilke fejl der kan opstå. *Det er en god ide at teste de enkelte komponenter i programmet, efterhånden som du laver dem.*

Du kan få et fingerpeg om, hvad vi vil sige til det output, dit program producerer, ved at skrive `DM507check`. Du skal, før du afgiver kommandoen, placere dig i det katalog, som alle dine oversatte `Java`-programmer ligger i. Så vil dit program blive kørt på nogle af testfilerne i `/home/IMADA/courses/dm507/Tests`. Bemærk, at testen kun indeholder nogle få testeksempler, så hvis der ikke findes fejl, betyder det ikke nødvendigvis, at dit program fungerer korrekt.

Det er en rigtig god ide at køre DM507check, inden du afleverer. På den måde sikrer du bl.a., at dit program bruger det korrekte input- og output-format.

Programmet `DM507check` kan afbrydes med `Ctrl-c`.

6 Hvilken algoritme er bedst?

I eksemplet i figur 1 bruger FF, NF og NFD 4/3 gange så mange bins som nødvendigt. Det samme er tilfældet for KNAPSACK i eksemplet sidst i afsnit 3.2.

Prøv at overveje, hvor langt de fem algoritmer hver især kan komme fra den optimale løsning. Dvs. prøv for hver algoritme, om du kan finde eksempler, hvor løsningen er mere end 4/3 gange fra det optimale.

Overvej også gerne, hvor langt algoritmerne kan komme fra den optimale løsning, hvis vi kun ser på "store" eksempler; f.eks. eksempler hvor der skal bruges mindst 10 bins.

7 Rapporten

Rapporten skal indeholde pseudokode for de fem algoritmer og gerne et klassediagram. Argumentér for, at algoritmerne har de ønskede køretider.

Rapporten skal desuden indeholde en udskrift af hele programmet. Denne udskrift skal være identisk med det elektronisk afleverede program. Der skal være en beskrivelse af de væsentligste valg, der er truffet i forbindelse med implementeringen, samt begrundelser herfor.

Beskriv de eksempler, du kom frem til under afsnit 6. Forklar gerne, hvordan du kom frem til disse eksempler, og om du tror, der findes endnu "værre" eksempler.

Endelig skal der være en overordnet beskrivelse af din teststrategi. Dvs. du skal *uden at referere til programmet* beskrive dine overvejelser i forbindelse med design af testen. Derudover skal selve testen dokumenteres, med reference til den overordnede teststrategi.

8 Aflevering

Programmet skal afleveres elektronisk. Den elektroniske aflevering foregår på følgende måde: Opret et katalog (directory), som indeholder alle dine `.java`-filer til opgaven *og intet andet*. Stil dig i dette katalog. Brug først `ls -la` for at sikre, at du står det rigtige sted, og at de rigtige filer er der. Afgiv derefter kommandoen `aflever DM507`.

D.v.s. gør følgende.

```
cd <opgave-katalog>
```

```
ls -la
aflever DM507
```
hvor `<opgave-katalog>` indeholder alle dine `.java`-filer til opgaven og intet andet.

Bemærk, at du (inden afleveringsfristen) kan aflevere flere gange. Kun den sidste aflevering tæller (de andre slettes).

Rapporten afleveres til forelæseren eller på IMADAs sekretariat. Denne opgaveformulering er vedhæftet en forside, som skal anvendes ved afleveringen.

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

9 Yderligere formalia

Bemærk, at den obligatoriske opgave er en del af eksamen, så reglerne for en eksamenssituation gælder også her. Dvs. du må ikke modtage hjælp fra andre, og du skal sørge for, at andre ikke kan læse dine filer. En god måde at beskytte sine filer på er følgende.

```
mkdir DM507projekt
chmod 700 DM507projekt
```

I må gerne snakke sammen om den overordnede løsning og lære af hinanden, men når I går i gang med at skrive ting ned, såvel program som rapport, skal I arbejde selvstændigt. Overtrædelse af dette er eksamenssnyd og behandles som sådan.

God arbejdslyst ☺

Institut for Matematik og Datalogi
Syddansk Universitet, Odense

4. oktober 2007
LMF

DM507 eksamen, Efteråret 2007
Obligatorisk Opgave

Skriv tydeligt (maskinskrift eller blokbogstaver)

Navn: .....

Fødselsdato: .....

Brugernavn (login): .....

Instruktor	Jacob	Mikkel
Sæt kryds		

Besvarelsen omfatter ..... nummererede sider.