Opgave 1 (20%)

Et binært træ med heltal i knuderne kan repræsenteres som en værdi af følgende rekursive type:

Type Tree = **Prod**(val: Int, left, right: Tree)

hvor det tomme træ angives som ?-Tree. Vi er interesserede i at afgøre, om et træ er et deltræ af et andet. Til det formål vil vi skrive en værdiprocedure:

Proc Deltræ[s, t: Tree] → (Bool)

der afgør, om s er et deltræ af t. En naiv algoritme for dette problem er, for hver knude i træet t, at afgøre om s er identisk med det deltræ af t, der har denne knude som rod. Det vil give en tidskompleksitet på $O(|t||s|)$.

En smartere algoritme er kun at undersøge dette for de knuder i t, hvis undertræ har det samme antal knuder som s. Man kan let se, at dette forbedrer tidskompleksiteten til $O(|t| + |s|)$.

> Implementer proceduren Deltræ, så den benytter den smarte algoritme. Der lægges vægt på, at besvarelsen er letlæselig, detaljeret og korrekt.

Opgave 2 (20%)

Det er velkendt, at følgende algoritme er gyldig og korrekt.

Algoritme: Heltalskvadratrod
Stimulans: n: $n \geq 0$
Respons: r: $r^2 \leq n < (r+1)^2$
Metode: r, h := 0, 1
do $\{ (0 \leq r) \wedge (r^2 \leq n) \wedge (h = (r+1)^2) \}$
  $h \leq n \rightarrow$ r, h := r+1, h+r+r+3
od

Man kan tilsvarende konstruere en algoritme, der beregner *heltalskubikroden* af et ikke-negativt tal.

Algoritme: Heltalskubikrod
Stimulans: n: $n \geq 0$
Respons: r: $r^3 \leq n < (r+1)^3$
Metode: ≪initialiser r, h og t≫
do $\{ (0 \leq r) \wedge (r^3 \leq n) \wedge (h = (r+1)^3) \wedge (t = (r+1)^2) \}$
  $h \leq n \rightarrow$ ≪opdater r, h og t≫
od

a) Udfyld de manglende stumper i algoritmen, således at opdateringen kun benytter additioner.

b) Bevis, at algoritmen fra a) er gyldig og korrekt.

Opgave 3 (20%)

I en vilkårlig uorienteret vægtet graf med n knuder og m kanter kan man som bekendt finde vægten af det letteste udspændende træ i tid $O((n+m)\log n)$ ved brug af Prims algoritme.

I denne opgave skal vi se, hvordan man kan udnytte egenskaberne ved nogle specielle grafer til at opnå forbedrede tidskompleksiteter. Det antages *ikke*, at grafen er sammenhængende; derfor betragter vi den letteste udspændende skov, der består af det letteste udspændende træ for hver sammenhængskomponent.

a) Antag, at vi kun ser på grafer, i hvilke alle kanter har vægten 1. Skitser en simpel og effektiv algoritme, der beregner vægten af den letteste udspændende skov. Argumentér for algoritmens tidskompleksitet.

b) Antag, at vi kun ser på grafer, i hvilke alle kanter har vægten 1 eller 2. Skitser en simpel og effektiv algoritme, der beregner vægten af den letteste udspændende skov. Argumentér for algoritmens tidskompleksitet.

c) Antag, at vi kun ser på grafer, i hvilke alle knuder har grad højst 2. Skitser en simpel og effektiv algoritme, der beregner vægten af den letteste udspændende skov. Argumentér for algoritmens tidskompleksitet.

Opgave 4 (20%)

På et gulv ønsker man at skrive sætninger ved hjælp af fliser med bogstaver på. Desværre kan man ikke nødvendigvis købe fliser med enkelte bogstaver på, så det kan være et problem at få skrevet de ønskede sætninger.

Generelt er vi givet en samling flisetyper, repræsenteret som en liste F af tekster, samt en sætning S, der blot er en tekst. Vi ønsker at afgøre, om man med (en ubegrænset forsyning af) de givne flisetyper kan skrive den valgte sætning.

For eksempel kan vi betragte sætningen:

```
DATALOGI ER BARE SÅ SJOVT
```

og følgende samling af flisetyper:

```
 BAR
A
Å SJO
ABE
ALO
DA
DAT
DATA
E
E S
ER
I ER
JO
KEDELIGT
LOG
LOGI
LOGO
R
R BA
VT
```

Her kan problemet løses på følgende måde:

`[D|A|T|A] [L|O|G] [I| |E|R] [ |B|A|R] [E| |S] [Å| |S|J|O] [V|T]`

Det generelle problem kan løses med følgende rekursive algoritme:

```
Proc Fliser[S: Text] (i, j: Int) → (Bool)
    if i = j → return true fi
    (+ Var k: Int
        k := 0
        do k< | F |  →
            if F.(k) = S(i..j) → return true fi
            k := k+1
        od
    +)
    (+ Var m: Int
        m := i+1
        do m<j →
            if Fliser[S](i, m) ∧ Fliser[S](m, j) → return true fi
            m := m+1
        od
    +)
    return false
end Fliser
```

a) Beskriv i ord hvorledes algoritmen fungerer og vis, at dens udførelsestid tilhører $\Omega(2^{|S|})$.

b) Benyt dynamisk programmering til at opnå en mere effektiv algoritme. Hvad bliver den forbedrede tidskompleksitet?

Opgave 5 (20%)

Der skal konstrueres en box Bush med følgende udseende:

> **Box** Bush
>   **Type** B = ≪sæk af heltal≫
>   **Proc** Init [b: B]
>   **Proc** Insert [b: B] (i: Int)
>   **Proc** Remove [b: B] (i: Int)
>   **Proc** Freq [b: B] (i: Int) → (Int)
>   **Proc** Select [b: B] (k: Int) → (Int)
> **end** Bush

som realiserer en datastruktur hvis værdier er *sække* af heltal. Init skaber en tom sæk, Insert og Remove henholdsvis indsætter og fjerner en forekomst af tallet i, Freq returnerer antallet af forekomster af tallet i, og endeligt returnerer Select det k'te element i sækken sorteret efter størrelse (startende med indeks 1).

For sækken:

$$< 0, 3, 3, 4, 4, 7, 8, 8, 8, 9, 11, 11, 11, 11, 13 >$$

vil Freq[b](4) således returnere værdien 2, medens Select[b](6) vil returnere værdien 7.

I det følgende angiver ||b|| antallet af *forskellige* elementer i sækken b, hvorimod |b| angiver det samlede antal elementer.

a) Beskriv en realisation af typen B, så Init får tidskompleksitet i $O(1)$ og de øvrige operationer får tidskompleksitet i $O(\log ||b||)$.

b) Angiv en algoritme, der kan sortere en liste S i tid $O(|S| \log(||S||))$ ved brug af en box Bush.