Eksamen i

Algoritmer og Datastrukturer

IAI 20102

Høgskolen i Østfold
Avdeling for informatikk
og automatisering

Tirsdag 3. desember 2002, kl. 09.00 - 14.00

LØSNINGSFORSLAG

Del 1 – Totalt 20%

alg1: En enkel løkke, går fra n og ned til 1, det gir lineær arbeidsmengde

alg2: To nestede løkker, ytre går fra 0 til m-1, den indre går fra 0 til n-1, der n er lik 42+m, dvs. at indre løkke går fra 0 til 41+m. Dermed kvadratisk med hensyn på m.

alg3: Algoritmen skriver ut første element tabellbiten den får inn, [a, b]. Deretter finner den midtpunktet i denne tabellen og behandler videre siste halvpart av tabellbiten. Hvis algoritmen første gang ble kalt med tabell med lengde 100, og med a=0 og b=99, ville vi få følgende utvikling:
0-99, 44-99, 71-99, 65-99, 82-99, ... etc. Dermed ser det ut til at denne metoden er logaritmisk, fordi vi halverer størrelsen på input i hvert kall.

Ser vi nærmere på hva som skjer når tabellen vil bli liten, skjer dette med eksempelet over:
96-99, 97-99, 98-99, 98-99, 98-99, ...altså terminere ikke algoritmen, men går i evig løkke. I slike arbeidstilfeller er ikke algoritmens orden udefinert (går mot uendelig).

Ser vi enda nærmere på problemet, og starter metoden med a >= b, vil denne metoden terminere umiddelbart, og vi får konstant orden. Hvis noen også har oppdaget dette, vil det selvsagt også gi full uttelling.

Summa summarum: De som argumentert i retning av minst en av disse forklaringene, og angitt orden ut fra det, får full uttelling.

alg4: To etterfølgende løkker. Den første går fra 0 til arr.length-1. Den andre går fra 0 til i-1, der i er lik arr.length-1. To løkker av lineær orden, som ikke er nestede, gir totalt lineær orden.

alg5: Tre nestede løkker. Ytre løkke går fra 0 til m-1. Midtre løkke teller seg nedover fra m-i til 1, der i er gitt av ytre løkke. Totalt sett er dette kvadratisk med hensyn på m. Da gjenstår det å kikke på indre løkke. Denne går nedover fra a-j til 1, der a=42, og j er gitt av midtre løkke. Dermed løper denne løkke et antall mellom 1 og 41...som jo gir en konstant arbeidsmengde. Hele algoritmen er dermed kvadratisk.

Slutt på del 1

Del 2 - 40%

Oppgave 2.1 - 5%

Et binærtre er enten tomt eller består av en rot samt et venstre subtre og et høyre subtre som begge er binærtrær.

I et binært søketre vil følgende holde for en vilkårlig node: Verdien i denne noden er større (eller like) alle verdiene i nodene i venstre subtre, og mindre (eller lik) alle verdiene i nodene i høyre subtre.

Oppgave 2.2 - 5%

```
int antallNoder(Node rot) {
  // Vi stopper ved en tomt subtre, som jo ikke har noen noen noder
  if (rot == null) return 0;
  // Vi bruker egenskapen at antallet noder i et (sub)tre er lik
  //noden selv pluss antall noder i venstre subtre pluss
  // antall noder i høyre subtre.
  return 1 + antallNoder(rot.venstre) + antallNoder(rot.høyre);
}
```

Oppgave 2.3 - 5%

Her må vi huske på at høyen er lik antall nivåer minus en.

```
int høyde(Node rot) {
  // Vi benytter oss av at høyden i et tre er lik
  // nivået vi er på (rota) + antall nivåer i subtreet (venstre/høyre)
  // med størst høyde

  // Vi må sørge for at bladnodene får høyde lik 0.
  if (rot == null) return -1;

  // Finner høyde i venstre/høyre subtre rekursivt
  int v = høyde(rot.venstre);
  int h = høyde(rot.høyre);

  //Den totale høyden er da lik 1 pluss høyden til det største subtreet
  if (v>h) {
    return v+1;
  }
  else {
    return h+1;
  }

}
```

Oppgave 2.4 - 5%

Her må vi huske på at lovlig verdi er avhengig av verdien på forfedrenodene.

```
boolean sjekk(Node rot) {
  // Rota i et BST kan ha alle mulige verdier, altså
  // i intervallet –uendelig til +uendelig
  return sjekk(rot, Integer.MIN_VALUE, Integer.MAX_VALUE);

}

boolean sjekk(Node rot, int min, int max) {
  // En null node er lovlig uansett
  if (rot == null) return true;

  // Hvis rota sin verdi er mindre enn minste lovlige verdi eller
  // større enn største lovlige verdi har vi et ulovlig subtre.
  if (rot.verdi < min || rot.verdi > max) return false;

  // Vi sjekker deretter rekursivt om venstre er lovlig.
  // Nodene i venstre subtre kan ikke være større enn
  // rotas verdi, men heller ikke mindre enn den minste lovlige
  // verdien for dette treet.
  if (!sjekk(rot.venstre, min, rot.verdi)) return false;

  // Vi sjekker deretter rekursivt om høyre er lovlig.
  // Nodene i høyre subtre kan ikke være mindre enn
  // rotas verdi, men heller ikke større enn den største lovlige
  // verdien for dette treet.
  if (!sjekk(rot.høyre, rot.verdi, max)) return false;

  // Hvis begge subtrær er lovlige, er også dette treet lovlig
  return true;
}
```

Oppgave 2.5 - 10%

Her bruker vi søkelogikken i et BST på referanseform, og oversetter slik at den fungerer på et tabellimplementert BST.

```
int finn(int[] tre, int verdi) {
  // Starter med å sjekke rota, dvs. første element i tabellen
  return finn(tre, 0, verdi);
}
```

```
int finn(int[] tre, int rot, int verdi) {
  // Hvis rotindeksen havner utenfor tabellen,
  // eller verdien i dette tabellelementet er N,
  // returnerer vi N, dvs. at verdien vi søker etter ikke finnes
  if (rot >= tre.length || tre[rot] == N) return N;

  // Hvis element nr. rot i tabellen tre er like verdien
  // vi søker etter, så returnerer vi denne indeksen
  if (tre[rot] == verdi) return rot;

  // Hvis verdien vi leter etter er mindre enn rotas verdi,
  // må vi lete i venstre subtre (tilsvarer å kalle søkerutinen
  // for venstre barn, som har indeks (rot*2)+1
  else if (tre[rot] > verdi) {
    return finn(tre, (rot*2)+1, verdi);
  }
  // ...ellers er verdien vi leter etter større enn rotas verdi,
  // og vi må lete i høyre subtre (tilsvarer å kalle søkerutinen
  // for høyre barn, som har indeks (rot*2)+2

  else {
    return finn(tre, (rot*2)+2, verdi);
  }
}
```

Oppgave 2.6 - 10%

Her gjelder det å traversere hele treet vårt (i vilkårlig orden, vi velger her preorder)
og legge hver nodes verdi i riktig tabellelement, gitt av formelen for indeks til venstre og høyre barn.

```
int[] konverterTilTabell(Node rot) {
  // Først lager vi en tabell av riktig størrelse
  int[] tabell = lagTabell(rot);

  // Deretter fyller vi tabellen med verdiene i treet
  tilTabell(rot, tabell, 0);
  // Og så returnerer vi den ferdige tretabellen.
  return tabell;
}

int[] lagTabell(Node rot) {
  // Lager en tabell av størrelse lik 2^h –1,
  // der h er lik antall nivåer, eller høyden pluss 1
  // Denne formelen gir oss antallet noder i et komplett binærtre,
  // som jo igjen tilsvarerer det maksimale antallet noder i et binærtre
  // Dermed er vil tabellen garanterert være stor nok

  // Alternativ kunne tabellen redimensjoneres etterhvert som
  // fylte den opp i den rekursive rutinen, eller vi kunne traversert
  // treet og talt opp hvor mange elementer vi trengte.

  // Finner antall nivåer ved å bruke høydemetoden
  int h = høyde(rot)+1;
  // Lager stor nok tabell basert på antall nivåer
  int[] tabell = new int[<2^h–1>],

  // Fyller tabellen med ”nullreferanser”
  for (int i = 0; i < tabell.length; i++) {
    tabell[i] = N;
```

```
  }

  return tabell;
}
```

Vi lar Node rot tilsvarer tabellelement nr. index:

```
void tilTabell(Node rot, int[] tabell, int index) {
  // Gjør ingen ting med nullnodene,
  // tabellen er på forhånd initialisert med ”nullreferanser”
  if (rot == null) return;

  // Settere tabellelement nr. index til verdien til rot
  tabell[index] = rot.verdi;

  // Indeksverdien til venstre barn er (index*2)+1)
  // og vi bygger venstre subtre rekursivt basert på denne verdien
  tilTabell(rot.venstre, tabell, (index*2)+1);

  // Indeksverdien til høyre barn er (index*2)+2)
  // og vi bygger høyre subtre rekursivt basert på denne verdien
  tilTabell(rot.høyre, tabell, (index*2)+2);
}
```

Slutt på del 2

Del 3 - 30%

Oppgave 3.1 - 5%

Husk at n*n kvadratet er lagt ut radvis. Da må vi finne startindeksen til rad nr. r., og summer de n påfølgende elementer, som jo representerer raden.

```
int sumRad(int[] kvadrat, int r) {
  int s = 0;
  // Tabellen er n*n, der n tilsvarer antall rader/kolonner
  int n = (int)Math.sqrt(kvadrat.length);

  // Finner starten på rad r.
  int start = r*n;
  // Summerer opp n elementer fra og med indeks start
  for (int i = 0; i < n; i++) {
    s += kvadrat[start+i];
  }
  return s;
}
```

Orden: O(n), der n er antallet kolonner/rader.

Oppgave 3.2 - 5%

```
boolean sjekkKvadrat(int[] kvadrat) {
      // Finner først summen av første rad
      // Alle de andre radene, kolonnene og diagonalene
      // må ha samme sum
     int magisk_sum = sumRad(kvadrat, 0);
     int n = (int)Math.sqrt(kvadrat.length);

      // Sjekker summen i de resterende n-1 radene
      // Hvis en sum avviker fra magisk_sum, bryter vi og
      // returnerer false
     for (int i = 1; i < n; i++) {
         if (sumRad(kvadrat, i) != magisk_sum) return false;
     }

      // Sjekker summen i alle kolonnene
      // Hvis en sum avviker fra magisk_sum, bryter vi og
      // returnerer false
     for (int j = 0; j < n; j++) {
         if (sumKolonne(kvadrat, j) != magisk_sum) return false;
     }

      // Sjekker summen i diagonalene
      // Hvis en sum avviker fra magisk_sum, bryter vi og
      // returnerer false
     if (sumDiag1(kvadrat) != magisk_sum) return false;
     if (sumDiag2(kvadrat) != magisk_sum) return false;

      // Hvis vi har kommet så langt, er det et magisk kvadrat!
     return true;
 }
```

Orden: Vi sjekker n antall rader, deretter n antall kolonner, så to diagonaler (altså i verste tilfelle, der vi faktisk har et magisk kvadrat). I hver av disse operasjonene summerer vi n antall elementer. Vi har da totalt 2n+2 antall summeringsmetoder som hver er av O(n). Totalt sett gir dette kvadratisk orden med hensyn på n, O(n*n).

Oppgave 3.3 - 10%

Det å finne alle kvadrater er ekvivalent med å finne alle permutasjoner av tallene 1, 2, ..., n*n. Vi bruker en standard permuasjonsalgoritme, sjekker hver gang vi har en ny full permutasjon om dette er et magisk kvadrat, skriver i så fall ut kvadratet, og teller opp.

```
int skrivAlleMagiskeKvadrater(int n) {
  int antall = 0;
  // Lager permutasjonstabell og tabell som holder rede på brukte elementer
  int[] perm = new int[n*n];
  boolean[] brukt = new boolean[n*n];
  for(int i = 0; i < brukt.length; i++) {
    brukt[i] = false;
  }

  // Starter med å permutere posision nr. 0
  return permutasjoner(perm, brukt, 0);
```

```
}

int permutasjoner (int[] perm, boolean[] brukt, int p) {
  // Hvis vi har en full permutasjon,
  // sjekker vi om vi har et magisk kvadrat
  if ( p == perm.length ) {
    if (sjekkKvadrat(perm)) {
      skrivKvadrat(perm);
      return 1;
    }
    // Hvis det ikke var magisk, returnerer vi 0.
    return 0;
  }

  // Nullstiller antall magiske kvadrater
  int no = 0;
  // Prøver med alle gjenværende lovlige verdier
  // og setter disse i posisjon p
  for(int = 0; i < perm.length; i++) {
    // Hvis ikke elementet er brukt, prøver vi med dette
    if(!brukt[i]) {
      // Husk at tallene våre går fra 1, og ikke fra 0
      perm[p] = i+1;
      // En permutasjon kan ikke inneholde identiske verdier
      brukt[i] = true;
      // Øker antallet hvis vi finner noen ved å
      // permutere resten av tabellen rekursivt
      no += permutasjoner(perm, brukt, p+1);
      // Backtracking!
      brukt[i] = false;
  }
  // Returnere alle magiske kvadrater som ble funnet
  return no;
}
```

Orden: Vi permuterer n*n antall elementer, dette gir oss O(n*n!). Selv om vi ved hver ny permutasjon må foreta en sjekk av kvadratisk orden, gir ikke dette totalt en høyere orden, men bare gir oss en høyere konstant faktor: Set m= n*n: O(m)*O(m!) = O(m*(m!)), som igjen er mindre enn O((m+1)!)...

Oppgave 3.4 - 10%

Bruke avskjæringer samt speilingsegenskaper...evt. formler (de finnes for visse tilfeller), ...eventuelt diverse smarte triks, sjekk nettet!

En naturlig måte å avskjære på, er å som følger: Vi observerer at kvadratet permuteres radvis, først bestemmes det første elementet i første rad, deretter andre element i første rad, ...siste element i første rad, første element i andre rad...osv. Hver gang vi får en ny permutasjon av første rad lar vi dette være vår magiske sum, som jo må stemme med alle påfølgende rad/kolonne/diagonalsummer. Videre sjekker vi hver gang vi har bestemt en posisjon som tilsvarer siste element i en rad, respektive kolonne og diagonal. Vi avbryter da videre permutering med en gang vi har en situasjon der vi vet noe har gått galt.

Dette gir en forbedring av effektivitet, men ikke orden. Prøv nedenstående implementasjon for n=3, deretter for n=4 :-)

Man får god uttelling med kun en utlegning liknende den ovenfor...implementering av en metode med fulle avskjæringer, evt. speilinger etc. er noe kun de færreste vil kunne få til på eksamen...men endel pseudokode burde være mulig...

Vi antar vi har en global variabel magisk_sum.

```
  public boolean finnEtMagiskKvadrat(int n) {
        int[] perm = new int[n*n];
        boolean[] brukt = new boolean[n*n];
        for(int i = 0; i < brukt.length; i++) {
            brukt[i] = false;
        }
        return permutasjonerII(perm, brukt, 0);
    }

    boolean permutasjonerII (int[] perm, boolean[] brukt, int p) {
        if ( p == perm.length ) {
       // Trenger ikke sjekke, er vi kommet så langt er kvadratet korrekt
            skrivKvadrat(perm);
            return true;
        }

        boolean ok = true;
        for(int i = 0; i < perm.length; i++) {
            ok = true;
            if(!brukt[i]) {
                perm[p] = i+1;
                int n = (int)Math.sqrt(perm.length);
                if (p == n-1) {
                     // Hvis vi har permutert ferdig første rad,
                     // resetter vi magisk sum
                    magisk_sum = sumRad(perm, p/n);
                }
                 // Hvis vi har ferdige andre rad
                else if (p >= (n*(n-1))-1) {
                   // Går kun videre hvis alle ferdig permuterte
                   // rader/kolonner/diagonaler er ok
                    ok = !feilRadSum(perm, p) && !feilKolonneSum(perm, p)
&& !feilDiagSum(perm, p);
                }

                if (ok) {
                     // Standard permutasjon med backtracking,
                     // avbryter hvis vi har funnet en løsning
                    brukt[i] = true;
                    if (permutasjonerII(perm, brukt, p+1)) {
                        return true;
                    }
                    brukt[i] = false;
                }
            }
        }
        return false;
    }

     // Sjekker om posisjon p tilsvarer siste element i en rad
     // Regner da ut radsum og sjekker mot magisk_sum
    boolean feilRadSum (int[] kvadrat, int p) {
        int n = (int)Math.sqrt(kvadrat.length);
        // Rad?
        if (p % n == n-1) {
            if (sumRad(kvadrat, p/n) != magisk_sum) {
                return true;
            }
```

```
    }
    return false;
}
```

```
  // Sjekker om posisjon p tilsvarer siste element i en kolonne
  // Regner da ut kolonnesum og sjekker mot magisk_sum
boolean feilKolonneSum (int[] kvadrat, int p) {
    int n = (int)Math.sqrt(kvadrat.length);
    // Kolonne?
    if (p >= n*(n-1)) {
        if (sumKolonne(kvadrat, p%n) != magisk_sum ) {
            return true;
        }
    }
    return false;
}

  // Sjekker om posisjon p tilsvarer siste element i en diagonal
  // Regner da ut diagonalsum og sjekker mot magisk_sum
boolean feilDiagSum (int[] kvadrat, int p) {
    int n = (int)Math.sqrt(kvadrat.length);
    // Diag1?
    if (p == kvadrat.length-1) {
        if (sumDiag1(kvadrat) != magisk_sum ) {
            return true;
        }
    }
    // Diag2?
    if (p == n*(n-1)) {
        if (sumDiag2(kvadrat) != magisk_sum ) {
            return true;
        }
    }
    return false;
}
```

Slutt på del 3

Vedlegg

De følgende klassene og metodene skal brukes i Oppgave 2. En del metoder er angitt som allerede implementerte, *disse skal altså ikke implementeres*, men kan brukes i andre metoder. Du står fritt i å legge til ekstra funksjonalitet i form av datamedlemmer og metoder. Husk at metodene som skal implementeres kan brukes i andre metoder selv om du ikke har klart å implementere de!

Oppgave 2: `class BST`

Klassen **`BST`** inneholder metoder knyttet til binære søketrær.

```
class BST {



    // Returnerer antall noder i binærtreet som er dannet av rot
  int antallNoder(Node rot) { /* Skal implementeres i 2.2! */ }


    // Returnerer høyden (dvs. antall nivåer-1) i binærtreet
    // som er dannet av rot
  int høyde(Node rot) { /* Skal implementeres i 2.3! */ }


    // Returnerer true om binærtreet som er dannet av rot
    // er et binært søketre, false ellers.
  boolean sjekk(Node rot) { /* Skal implementeres i 2.4! */  }


    // Vi antar at tre er et binært søketre på tabellform
    // Metoden returnerer indeksen som tilsvarer tallet verdi.
    // Hvis en slik verdi ikke finnes, returnes konstanten N.
  int finn(int[] tre, int verdi) { /* Skal implementeres i 2.5! */ }


    // Metoden konverterer et binærtre til tabellform
    // Tabellen returneres
  int[] konverterTilTabell(Node rot) { /* Skal implementeres i 2.6! */ }


  ...

}


  // Binærtrenode
class       Node {
  ...
  Node      venstre; // venstre barn
  Node      høyre;   // høyre barn

  int       verdi;
}
```

Oppgave 3: `class MagiskKvadrat`

Klassen **`MagiskKvadrat`** inneholder metoder knyttet til magiske kvadrater.

```
class MagiskKvadrat {


    // Returnerer summen av elementene i rad nr. r.
    // Du kan regne med at r ligger i intervallet [0,n-1],
    // der n er størrelsen på kvadratet.
  int sumRad(int[] kvadrat, int r) { /* Skal implementeres i 3.1! */  }


    // Returnerer summen av kolonne nr. k
    // Kan med fordel benyttes i oppgave 3.2.
    // Metoden er å anse for ferdig implementert og klar til bruk.
  int sumKolonne(int[] kvadrat, int k) { /* Ferdig implementert */  }


    // Returnerer summen av diagonalen
    // øverste venstre hjørne til nederste høyre.
    // Kan med fordel benyttes i oppgave 3.2.
  int sumDiag1(int[] kvadrat) { /* Ferdig implementert */  }


    // Returnerer summen av diagonalen
    // nederste venstre hjørne til øverste høyre.
    // Kan med fordel benyttes i oppgave 3.2.
  int sumDiag2(int[] kvadrat) { /* Ferdig implementert */  }


    // Returnerer true hvis kvadratet er magisk, ellers false.
  boolean sjekkKvadrat(int[] kvadrat) { /* Skal implementeres i 3.2! */ }


    // Skriver ut et kvadrat
    // Kan med fordel benyttes i oppgave 3.3.
  void skrivKvadrat(int[] kvadrat) { /* Ferdig implementert */ }


    // Returnerer antallet magiske kvadrater av størrelse n*n.
    // Hvert magisk kvadrat skrives ut.
    // Metoden skal generere alle mulige kvadrater av denne størrelse,
    // og sjekke hvert enkelt om det er magisk.
  int skrivAlleMagiskeKvadrater(int n) { /* Skal implementeres i 3.3! */ }


    // Returnerer true hvis et magisk kvadrat
    // av størrelse n*n finnes, false ellers.
    // Metoden skal være så effektiv som mulig.
  boolean finnEtMagiskKvadrat(int n) { /* Skal implementeres i 3.4! */ }

  ...
}
```

Slutt på oppgavesettet