Universitetet i Bergen
Det matematisk-naturvitenskapelige fakultet

Bokmål

Eksamen i	:	I-120 Algoritmer, Datastrukturer og Programmering
Dato	:	11 Desember 2002
Tid	:	9:00 – 15:00
Antall sider	:	3 (+ vedlegg)
Tillatte hjelpemidler	:	Alle trykte og skrevne

- Du kan bruke alle `interface` og data typer som ble gjennomgått i løpet av kurset – bruk i størst mulig grad `interface` og unngå unødvendig bruk av konkrete data typer.

  (`interface Stack`, forskjellige `Iterator`, samt relevante deler av `interface BinaryTree` og `Graph` er gitt i vedlegg.)
- Du kan alltid anta at du har gitt implementasjoner av `interface` som brukes i programmet.
- Formuler eksplisitt alle ekstra forutsetningene du gjør og som ikke står i oppgaveteksten.
- Har du ikke tid til å skrive fullstendig program, kan presis pseudo-kode med tilstrekkelig forklaringer telle omtrent like mye.
- Klarer du ikke å svare på en (del)oppgave, kan du fortsette og anta i videre løsning at resultatet er alikevel tilgjengelig.
- Prosentsatsene ved hver (del)oppgave angir omtrentlig vektlegging ved sensur (og *ikke nødvendigvis* forventet arbeidsmengde).

1 Stabel (10%)

Vi har noen implementasjon(er) av `interface SvartHvit { boolean erHvit();}`. Skriv en metode `boolean equalNumber(ObjectIterator I, Stack S)` som bruker *kun én* stabel (aktuell parameter `S` ved metodekall antas å være en tom stabel) og som returnerer `true` hvis og bare hvis parameter iterator `I` inneholder samme antall av objekter med `erHvit()==true` som med `erHvit()==false`. (`I` inneholder en serie (av ukjent lengde) av kun `SvartHvit` objekter, der de som `erHvit()` og de som ikke er det kan komme om hverandre i vilkårlig rekkefølge.)

2 Binære trær (50%)

I denne oppgaven har du gitt (noen implemenatsjoner av) `interface Binary Tree` (se vedlegg) som i tillegg oppfyller noen datainvariant – i deloppgave 2.1 er det invarianten for binære søketrær mens i 2.2 for haug.

Vi antar at nøkler er heltall med vanlig ordning/sammenlikning og at de er lagret i `element()` attributtet av `Position`'er i treet. Der det er nødvendig, kan du anta at `int k` blir omdannet til passende nøkkelobjekt, f.eks., `Integer K`.

2.1 Søketrær (25%)

1. Formuler invarianten/betingelsen som et binært tre må oppfylle for å være et søketre, BST.
2. Tegn et BST som vil resultere fra innsetting av elementer med følgende nøkler: 5, 3, 8, 6, 9, 7, 1, 4 (i denne rekkefølgen, dvs. først 5, så 3, så 8, osv.). Innsetting sørger kun for søkeinvarianten og foretar ikke noen balanserning av treet.
3. Tegn så BST som man får ved å fjerne elementet med nøkkel 5 fra treet du fikk over.
4. Programmer en metode `void less(BinaryTree T, int x)` som skriver ut alle nøkler (heltall) samlet i et binært søketre `T` og som er mindre eller lik `x` (alle `k` slik at `k<=x`).

5. Hvilke tall – og *i hvilken rekkefølge* – blir skrevet ut hvis vi kaller din metode `less(T,7)`, der `T` er treet du fikk i pkt. 2?

6. Hva er verste fall kompleksitet av din metode (uttrykt som en funksjon av n – antall objekter lagret i treet)?

2.2 Haug (25%)

1. Formuler invarianten som et gitt binært tre må oppfylle for å være en haug (heap).

2. Tegn en haug som vil resultere fra innsetting av elementer med følgende nøkler: 7, 5, 9, 3, 8, 7, 1 (i denne rekkefølgen, dvs. først 7, så 5, så 9, osv.).

3. Tegn så en haug som man får etter å ha fjernet minste elementet fra haugen du fikk i pkt. 2.

4. Programmer en metode `void less(BinaryTree T, int x)` som skriver ut alle nøkler (heltall) samlet i et binært tre `T` – som er en haug – og som er mindre eller lik `x` (alle `k` slik at `k<=x`).

   Metoden skal ha kompleksitet $\mathcal{O}(m)$, der m er antall elementer med nøkkel `k<=x`.

5. Hvilke tall – og *i hvilken rekkefølge* – blir skrevet ut hvis vi kaller din metode `less(T,7)`, der `T` er treet du fikk i punkt 2 av denne oppgaven?

3 Grafer og generisk programmering (40%)

Vi betrakter rettede grafer, og ser på tre variasjoner av selve grafen:

i. Kanter har ingen attributter;

ii. Kanter har et heltallsattributt vekt – $vekt \geq 0$;

iii. Kanter har et heltallsattributt farge – $1 \leq farge \leq F$, der F er totalt antall mulige farger.

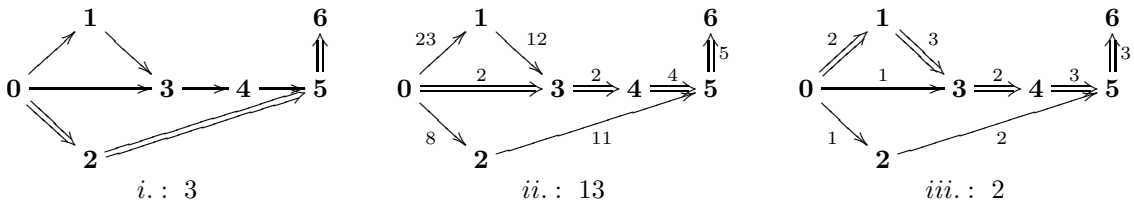
Gitt en rettet graf $G = (V, E)$, og noder $a, b \in V$, avhengig av grafvarianten over, kan vi stille essensielt samme spørsmål i tre forskjellige utgaver:

i. Hva er (stien med) færrest antall kanter fra a til b?

ii. Hva er (stien med) minst total vekt fra a til b?

iii. Hva er (stien med) færrest antall farger fra a til b?

Under har vi et eksempel på tre varianter av en graf – i. uten noen attributter, ii. med vekter, og iii. med farger – hver med et riktig svar (merket med doble kanter og tallet under) på tilsvarende spørsmål for noder $a = \mathbf{0}, b = \mathbf{6}$:



i. : 3 *ii.* : 13 *iii.* : 2

Oppgaven går ut på å programmere *én generisk* metode med pasende parametre som tillatter å svare på alle disse tre spørsmålene når kalt med respektiv grafvariant og passende implementasjon av parameter interface. (Man kan bruke en av algoritmene som ble gjennomgått på forelesninger – se Hint etter siste deloppgave.)

3.1 (30%)

1. Deklarer først en passende `interface Sml` som abstraherer felles trekk av de to første tilfellene, dvs. i. rettede grafer der kanter ikke har noen attributter, samt ii. de der kanter har heltallsvekter. Forklart kort meningen med hver metode. (`Sml` skal helst kunne brukes også for det tredje tilfelle, iii., men i denne deloppgaven får du uttelling selv om den ikke kan det.)

2. Skriv så en metode

   `int SSSP(Graph G, Node a, Node b, Sml V)`

   som skal fungere som en generisk metode for de to tilfellene. Som resultatet krever vi ikke en sti (som antydet på tegningen over) men kun heltallsverdi tilsvarende en slik sti (3 for grafen i. og 13 for grafen ii.).

   [Det er korrekthet og det generiske aspektet som er viktig her (se også deloppgave 3.2 under). Du trenger ikke å ta mye hensyn til effektivitet.]

3. Gi to implementasjoner:

   i. `class AntallSml implements Sml`, samt

   ii. `class VektSml implements Sml`

   slik at et kall

   i. `SSSP(G,a,b,new AntallSml())` på en gitt graf `G` uten vekter, skal gi riktig svar på spørsmål i. – antall kanter på korteste sti fra `a` til `b`, mens

   ii. `SSSP(G,a,b,new VektSml())` på en gitt graf `G` med vektede kanter, skal gi riktig svar på spørsmål ii. – vekten/lengden av korteste sti fra `a` til `b`.

Du vil måtte merke noder/kanter i grafen med ekstra verdier og vil dermed få bruk for det faktum at de utivder `interface Decorable`. Du kan da anta at objekter som identifiserer passende attributter er gitt (f.eks. `DD, vekt`, el.l.) men marker slike antakelser der de brukes.

3.2 (10%)

Programmer det som trengs for å håndtere også det tredje tilfelle iii av grafer med fargede kanter. Din metode fra punkt 2 i deloppgave 3.1 burde forbli uforandret – svaret skal nå bli det minimale antallet farger på kanter som danner en sti fra `a` til `b`. Hovedtyngden vil ligge i en passende implementasjon `class FargerSml implements Sml`. Forklar først hovedideen i din løsning. (Presis og godt forklart pseudo-kode kan gi omtrent like mye uttelling som detaljert program.)

[Hint: I eksempelet iii. over, legg merke til at man ikke kan anta at løsningen konstrueres inkrementelt slik det gjøres, f.eks. i Dijkstras algoritme. En løsning for noder **0**-**5**, kunne være **0**-**2**-**5**, men løsning for noder **0**-**6** er ikke en rett fram utvidelse av denne.

Man må altså huske, for hver node, alle mulige stier som fører til denne fra startnoden, og i hver iterasjon av algoritmen må man ta hensyn til (de relevante blant) alle slike muligheter.

Du kan gjøre en del forenklende forutsetninger der du trenger det, f.eks. at du har metoder som konverterer entydig noder og/eller kanter til noen tall, o.l. Uansett hva du forutsetter, gjør det klart og eksplisitt i din besvarelse.]

Michał Walicki Arild Waaler

Solution – prob. 1

We push new object (from the iterator) on the stack if either the stack is empty or else the top object is of the same kind as the one we are trying to push (isWhite()==true for both or for neither). If the two are of different kind, we merely pop the top of the stack. The answer is 'equal' if on termination the stack is empty.

```
boolean equalNumber(ObjectIterator I, Stack S) {
   BlackWhite o;
   while (I.hasMore()) {
     o = (BlackWhite)I.nextElement();
     if (S.isEmpty()) S.push(o);
     else if ( ((BlackWhite)S.top()).isWhite() ) {
       if (o.isWhite()) S.push(o);
       else S.pop(); }
     else {
       if (!o.isWhite()) S.push(o);
       else S.pop(); }
   return S.isEmpty();
}
```

Solution – prob. 2

2.1. BST

1. for each node: all keys in the left subtree are not-greater, and all in the right subtree are not-smaller than the node's key

2.-3. The tree for 5, 3, 8, 6, 9, 7, 1, 4 will be as on the left, and after removal of 5 as on the right:

4. `void less(BinaryTree T, int x) { less(T, T.root(), x); }`

```
void less(BinaryTree T, Position v, int x) {
   if (v != nil) {
     if (((Integer)v.element()).intValue() <= x) {
       print(v.element());
       less(T, T.leftChild(v), x);
       less(T, T.rightChild(v), x); }
     else less(T, T.leftchild(v), x);
} }
```

5. This question concerns in-, pre- or post-order. We made it pre-order, so our result will be: 5,3,1,4,6,7.

6. $\mathcal{O}(n)$, when the tree is unbalanced, essentially a linear structure (generally: $\mathcal{O}(heigh(T))$).

2.2. Heap

1. each node $\leq$ all its children

2.-3. the tree will be as on the left, and after removal as on the right:

4. `void less(BinaryTree T, int x) { less(T, T.root(), x); }`

```
void less(BinaryTree T, Position v, int x) {
   if (v != nil) {
     if (((Integer)v.element()).intValue() <= x) {
       print(v.element());
       less(T, T.leftChild(v), x);
       less(T, T.rightChild(v), x); }
} }
```

Here we can terminate the recursion at once we find something bigger than x, since in a heap everything below will be biger, too. We get $\mathcal{O}(m)$, since we write everything until we reach this condition, as all smaller than... is stored in the "upper part" of the heap.

5. Again, this is pre-order, so we will get: 1,5,7,3,7.

Solution – prob. 3

The first part could be obtained by a generalization of Dijkstra but, as sugggested in the hint, the second could not. We use Ford-Bellmann from start.

3.1. Number of edges and Length

1. ```
   public interface Sml {
      /* for initializing the X-attribute in the node v with MAX_VALUE */
      void initMax(Vertex v,Object X);
      /* for initializing (the starting) v with minimal value */
      void initMin(Vertex v,Object X);
      /* returns the value of the test for update,
        wrt.  the X-atribute, e.g.  X[c]+e.weight<X[d] */
      boolean less(Vertex c,Vertex d,Edge e,Object X);
      /* performs the update, if the test was true */
      void update(Vertex c,Vertex d,Edge e,Object X);
      /* converts the value stored in the X-atribute of v to int */
      int intValue(Decorable v,Object X);        }
   ```

2. Assume DD is a globally available Object identifying the respective atribute in the Vertex (which extends Decorable interface). Ignored all the exceptions which the metods from the given interface may rise.

   ```
   int SSSP(Graph G, Node a, Node b, Sml V) {
      EdgeIterator edges; Vertex v,c,d; Edge e;
      int n= G.numVertices();
      VertexIterator nodes= G.vertices();
      while (nodes.hasNext()) {
        v= nodes.nextVertex();
        V.initMax(v,DD); }
      V.initMin(a,DD);
      for (int i=1;i<n;i++) {
        edges= G.directedEdges(); // .edges() should be the same as G is directed
        while (edges.hasNext()) {
          e= edges.nextEdge();
          c= G.origin(e); d= G.destination(e);
          if (V.less(c,d,e,DD)) V.update(c,d,e,DD);
        }
      }
      return V.intValue(b,DD);
   }
   ```

3. ```
   class AntallSml implements Sml {
      public void initMax(Vertex v,Object X) {v.set(X,new Integer(Integer.MAX_VALUE);}
      public void initMin(Vertex v,Object X) { v.set(X,new Integer(0)); }
      public boolean less(Vertex c,Vertex d,Edge e,Object X) {
        if (intValue(c,X)==Integer.MAX_VALUE) return false;
        // this case is necessary to avoid overflow (wrap around for int in Java)
        else return intValue(c,X)+1 < intValue(d,X) ; }
      public void update(Vertex c,Vertex d,Edge e,Object X) {
        d.set(X, new Integer(intValue(c,X)+1); }
      public int intValue(Decorable v,Object X) {
        return (Integer(v.get(X))).intValue(); }
     }
   ```

Assume that also Edge uses the DD-atribute for storing weight:

```
class VektSml implements Sml {
   public void initMax(Vertex v,Object X) {
     v.set(X,new Integer(Integer.MAX_VALUE)); }
   public void initMin(Vertex v,Object X) { v.set(X,new Integer(0)); }
   public boolean less(Vertex c,Vertex d,Edge e,Object X) {
     if (intValue(c,X)==Integer.MAX_VALUE) return false;
     else return (intValue(c,X)+intValue(e,X) < intValue(d,X)); }
   public void update(Vertex c,Vertex d,Edge e,Object X) {
     d.set(DD, new Integer( intValue(c,X)+intValue(e,X) ); }
   public int intValue(Decorable v,Object X) {
     return (Integer(v.get(X))).intValue(); }
}
```

3.2. Colors

We do what the Hint suggests. The DD for a node v will hold the array $From$, with indices corresponding to the incoming edges. I.e., we assume given 1-1 functions between

$$\begin{array}{lll} edgeNo: & G.incidentEdges(v, EdgeDirection.IN) & \longrightarrow \quad [0...k] \\ & G.incidentEdges(v, EdgeDirection.IN) & \longleftarrow \quad [0...k] \times v \quad : edgeOf \end{array} \tag{1}$$

where $k+1$ is the number of incoming edges. Then, $From[i]$ is a sequence with objects corresponding to all the paths leading to v with the last edge being $i - v$.

Each path to v is identified with the colors it uses, i.e., is itself a set (sequence) of numbers $\leq F$. We represent it by, first, associating each color $1 \leq f \leq F$, with the $(f+1)$-th prime number $prim(f)$, and then the set of colors $S = \{f_1, f_2, ..., f_n\}$ (with $f_i \neq f_j$ for $i \neq j$) is simply the number $prim(S) = prim(f_1) \cdot prim(f_2) \cdot ... \cdot prim(f_n)$. (We take $(f+1)$-th prime number to avoid using 1, i.e., the first color $f = 1$ gets the number 2, the second 3, the third 5, etc.) Membership is then given by:

$$f \in S \iff prim(S) \; mod \; prim(f) = 0. \tag{2}$$

For convenience, we also let each v keep the sequence $paths$ of all paths leading to it, i.e., a kind of flattening of the whole $From$. (This is the invariant which makes a lot things simpler.)

The initial value (corresponding also to the MAX_VALUE), for each node, is array where $From[i].isEmpty()$ for all i, i.e., $paths.isEmpty()$. Filling in happens when, considering the edge $i - v$, the $From$ array in i is less than MAX_VALUE (has some paths) and $From[i]$ in v has fewer elements than the total number of paths in $From$ in i, i.e.,

$$!i.paths.isEmpty() \;\; \&\& \;\; v.From[i].size() < i.paths.size() \tag{3}$$

The latter means that some new paths leading to i have appeared since we considered the edge $i - v$ last time. We keep also the invariant that later appearing paths are put at the end of the $paths$. Copying the new paths to i and extending them with $i - v$ color for putting in $v.From[i]$, will thus concern only the $i.paths.size() - v.From[i].size()$ number of the paths from the end of $i.paths$.

We assume that in the environment where the following class is defined, we have access to the Graph G, the attribute Object DD, and the number of colors used by the G as well as the corresponding prim-function (other functions come from class Info on the following page):

```
class FargerSml implements Sml {
   private Graph G; private Object Attr; private int noF;
   public FargerSml(Graph Gr,Object X,int F) { G=Gr; Attr=X; noF=F;}
   public void initMax(Vertex v,Object X) {
     int no= 0;
     EdgeIterator ei= G.incidentEdges(v,EdgeDirection.IN);
     while (ei.hasNext()) { ei.nextEdge(); no= no+1; }
     v.set(X,new Info(no,v,Attr,noF)); }

// Assume that initMin is called on v after initMax and before any other updates
// i.e., v.From[i].isEmpty() for all i and v.paths.isEmpty() */
   public void initMin(Vertex v,Object X) { of(v,X).initOne(); }

   public boolean less(Vertex c,Vertex d,Edge e,Object X) {
     return (!of(c,X).isEmpty() &&
              of(c,X).paths.size() > of(d,X).sizeOf(edgeNo(e))); }

// Assume that e:c->d and the conversion function edgeNo, as explained in (1).
   public void update(Vertex c,Vertex d,Edge e,Object X) {
     of(d,X).add(edgeNo(e),of(c,X).paths); }

   public int intValue(Decorable v,Object X) { return of((Vertex)v,X).intValue();}
   private Info of(Vertex v, Object X) { return (Info)v.get(X); }
} // end FargerSml
```

```
public class Info { // objects to be stored in the DD-attribute for each node v
   private Vertex v; // vertex for which this is the Info
   private int k; // number of incoming edges to v
   private int noF; // number of colors (assume given prim-function)
   private Sequence[] From; // will have k entries
   private Object Attr; // the attribute to work on (could be more dynamic)
   protected Sequence paths= new SequenceImpl();

   public Info(int i,Vertex u,Object X,int F) {
     k= i; From= new Sequence[k]; v= u; Attr= X; noF= f;
     for (int z=0;z<k;z++) From[z]= new SequenceImpl(); }
   private int intValue(Decorable d,Object X) {
     return ((Integer)d.get(X)).intValue(); }
   private int intObj(Object d) { return ((Integer)d).intValue(); }

   public void add(int i,Sequence pth) {
// extend From[i] with ``last'' elements from pth (adding i-v);
// assuming:  pth.size() >= From[i].size(), and !pth.isEmpty()
   int seq; Position q,p;
     q= pth.first();
     if (!From[i].isEmpty()) { //iterate its size through pth
       p= From[i].first();
       while (p != From[i].last()) { p= From[i].after(p); q= pth.after(q); }
       q= pth.after(q); }
// now copy and extend the rest of pth:  add the color of i-v, if needed
     int col= prim(color(i)); // color gives int stored in Attr
     while (!pth.isLast(q)) { // each element/int in pth represents a set of colors
       seq= intObj(q.element()); // the set stored at q
       if (seq%col != 0) seq= seq*col; // add col if not in seq
       From[i].insertLast(new Integer(seq));
       paths.insertLast(new Integer(seq));
       q= pth.after(q); }
     seq= intObj(q.element()); // the sequence stored at the last Position
     if (seq% col != 0) seq= seq*col;
     From[i].insertLast(new Integer(seq));
     paths.insertLast(new Integer(seq));
   } // end add(...)

   public int intValue() { // the minimal number of colors at any From[i]
     int mi= 0; int seq,no;
     for (int i=0;i<k;i++) {
       ObjectIterator ith= From[i].elements();
       while (ith.hasNext()) {
         seq= intObj(ith.nextObject()); no= 0;
         for (int k=1;k<=noF;k++) { if (seq%prim(k) != 0) no= no+1; }
         if (no<mi) mi= no;
     } }
     return mi;
   } // end intValue()

   public int sizeOf(int i) { return From[i].size(); }
   public boolean isEmpty() { return paths.isEmpty(); }
   public void initOne() { paths.insertFirst(new Integer(1)) }
   private int color(int i) { // Edge stores color in Attr; edgeOf - see (1)
     return ((Integer)edgeOf(i,v).get(Attr)).intValue(); }
} // end Info
```