- Du kan bruke alle `interface` og data typer som ble gjennomgått i løpet av kurset – bruk i størst mulig grad `interface`'r og unngå unødvendig bruk av konkrete data typer.
- Der du er bedt om å programmere noe, skal du skrive en fullstendig Java kode. Dersom programmet ditt trenger en *implementasjon* av en gitt `interface`, kan du anta at en slik implementasjon er tilgjengelig uten å implementere den selv. Indiker steder der du bruker en slik antakelse.
- Kompleksitetskravene er gitt *relativt* til eventuell implementasjon av interface(ne) som brukes i programmet (dvs. der relevante operasjoner fra interface-implementasjon antas å være av orden $\mathcal{O}(1)$.)
- Dokumenter all kode (beskriv forutsetninger om parametere samt resultat av metoder, spesifiser invarianter for viktige løkker og rekursjon, etc.).
- Les hele oppgavesettet før du begynner å løse enkelte oppgaver.
- Prosentsatsene ved de enkelte oppgaver angir estimert tidsforbruk ved løsning samt omtrentlig vekt ved sensur.

1 Algoritmer – Fargelegging av kart (30%)

Vi har et kart over forskjellige land og oppgaven går ut på å fargelegge kartet slik at ingen to naboland (som har felles grense) får samme farge. Kartet er slik at:

- ingen land har kolonier (hvert land er en sammenhengdende region)
- det finnes ingen hav (kun landene som skal fargelegges)
- hvert land grenser mot minst étt annet land (ingen "isolerte" land).

Et kjent matematisk teorem sier at ethvert slik kart kan fargelegges med *høyst 4 farger*. Du skal designe en algoritme som fargelegger et kart gitt som input med høyst 4 farger.

1.1

Deklarer datastrukturen som du skal bruke for å løse fargeleggingsproblemet.

1.2

Design og forklar algoritmen som skal fargelegget et kart *med høyst 4 farger*. Algoritmen *skal* finne en lovlig fargelegging. (Vi antar, som sagt, at en slik alltid finnes.) Hva er kompleksiteten? Programmer så algoritmen.

2 Datastrukturer – Morsekode (40%)

Morsekoden bruker to symboler: prikk ($\bullet$) og strek ($-$), for å kode alle bokstaver (og en del andre tegn). Under er det gitt en del av Morsekode for latinske bokstaver.

$A : \bullet -$	$J : \bullet - - -$	$S : \bullet \bullet \bullet$
$B : - \bullet \bullet \bullet$	$K : - \bullet -$	$T : -$
$C : - \bullet - \bullet$	$L : \bullet - \bullet \bullet$	$U : \bullet \bullet -$
$D : - \bullet \bullet$	$M : - -$	$V : \bullet \bullet \bullet -$
$E : \bullet$	$N : - \bullet$	$W : \bullet - -$
$F : \bullet \bullet - \bullet$	$O : - - -$	$X : - \bullet \bullet -$
$G : - - \bullet$	$P : \bullet - - \bullet$	$Y : - \bullet - -$
$H : \bullet \bullet \bullet \bullet$	$Q : - - \bullet -$	$Z : - - \bullet \bullet$
$I : \bullet \bullet$	$R : \bullet - \bullet$	

F.eks., vil ordet "set" kodes som en streng "... . −" og ordet "cat" som "−. − . . − −". Vi bruker et enkelt mellomrom for å skille koding av to bokstaver og et dobbelt mellomrom for å skille koding av forskjellige ord.

Vi skal lage algoritmer for koding og dekoding av Morsemeldinger. Selv om vi bruker det latinske alfabetet, kan man generelt kode et vilkårlig alfabet v.hj.a. Morse. Tallet D angir størrelse på alfabetet – i eksemplet over er $D = 26$.

2.1 Dekoding

Vi skal først tenke på dekoding av enkle bokstaver og så av enkle ord. Input til algoritme er en Morsekode tilsvarende en enkel bokstav (eller, muligens, ingen bokstav dersom koden ikke er korrekt). Lengden av denne er n = antall prikker-og/eller-streker. (For koden av "S" er $n = 3$, for koden av "T" er den $n = 1$.)

Oppgaven går ut på å designe en $\mathcal{O}(n)$ algoritme for å dekode denne strengen – dvs. produsere en tilsvarende bokstav eller rapportere feil. Kompleksitetetn av algoritmen skal avhenge kun av n men ikke av D!

1. Velg en passende datastruktur – bruk helst (no)en passende `interface`.
2. Vis (tegn) delen av datastrukturen som representerer bokstavene A,E,I,U,N,M,S,T.
3. Programmer en algoritm som tar en streng `ms` som argument (denne skal inneholde en streng av priker-og/eller-streker tilsv. en Morse kode for en bokstav) returnerer den kodete bokstav (eller rapporterer feil)
4. Orden av algoritmen over skal være $\mathcal{O}(n)$ og uavhengig av alfabet-størrelsen D – begrunn at din algoritme tilfredstiller dette kravet. (For å se om det er tilfelle, sjekk at kjøretiden til din algoritme på *samme input* ikke ville forandres dersom man nå utvidet alfabetet (til, si $5 * D$ bokstaver) og lagt til nye koder for de nye bokstavene.)
5. Endelig, skal du lage en algoritme som tar som input en streng tilsv. et helt ord kodet i Morse. Programmer en algoritme som dekoder et slikt ord, dvs. som tar en streng med kodet ord som argument og returnerer en streng med ordet i klartekst form (eller rapporterer feil i koding). Du kan anta at argument-strengen inneholder ingen to etterfølgende mellomrom. Hva er kompleksiteten til denne?

2.2 Koding

Nå har vi en omvendt problemstilling: vi har en klartekst, for enkelhets skyld bare ett ord T, som skal kodes som en Morsekode. Vi lar t være lengden av T, dvs. antall bokstaver i ordet T. (For $T =$"set" er $t = 3$.)

Du skal designe en $\mathcal{O}(t * log(D))$ algoritme for koding av T.

1. Se om du kan bruke datastrukturen fra forrige deloppgave for dette formål – begrunn hvorfor eller hvorfor ikke.
2. Var den tidligere datastrukturen utilstrekkelig, velg en passende datastruktur.
3. Programmer en $\mathcal{O}(log(D))$ algoritme som koder en enkel bokstav gitt som input. (Nevn eksplisitt ev. tilleggsforutsetninger!)
4. Programmer så en algoritme som koder et helt ord T av lengde t, og er av orden $\mathcal{O}(t * log(D))$.

3 Programmering – Multigrafer (30%)

Vi skal se på en datatype `Multigraph`. Denne vil benytte seg av en datatype `Bag` som vi ser på først.

3.1

En *'bag'* er en uordnet samling der ett og samme element kan forekomme flere ganger. (En *'mengde'* er et spesial tilfelle der hvert element forekommer høyst én gang.) Vi har altså følgende abstrakte data type:

```
public interface Bag {
  Enumeration elements();
  int multipl(Object o); }
```

Den første metoden returnerer enumerering av alle elementene der hvert forekommer bare én gang. Den andre metoden returnerer antall forekomster av argument objektet `o` i bagen. F.eks. for bag `bg = {a,a,b,a,c}` vil `bg.elements()=={a,b,c}`, `bg.multipl(a)==3` mens `bg.multipl(d)==0`.

Mer presist: vi antar at objektene samlet i en bag har definert metode `equals`; metoden `multipl(o)` returnerer antall forekomster av objekter `x` slik at `o.equals(x)==true`.

Definer en metode

```
boolean erLike(Bag a, Bag b)
```

som avgjør om de to bagene er like, dvs. inneholder det samme antallet av de samme elementene.

Multigraf

En rettet *multigraf* er en graf der mellom to noder kan det være flere enn bare en (rettet) kant. Vi har følgende abstrakt data type:

```
public interface MultiDiGraph extends DiGraph {
  Bag kanter();
  Bag innKanter(Position v);
  Bag utKanter(Position v);
  Bag innNaboer(Position v);
  Bag utNaboer(Position v); }
```

Disse kommer i tillegg til metodene fra `DiGraph`. F.eks. `DiGraph` metoden `outAdjacentV(v)` vil, for en multigraf, returnere en enumerering av node-posisjoner `x` slik at det finnes (minst) en kant `v` $\rightarrow$ `x`. Metoden `utNaboer(v)` vil returnere en bag av slike node-posisjoner, dvs. i tillegg til selve enumerering av disse nodene, også multiplisitet av kanter fra `v` til `x` ("hvor mange ganger `x` er `v`'s ut-nabo") – her betrakter vi to kanter som like (`equals`) dersom de har den samme kilde- og målnode. (Man kan tenke seg at den første er implementert som `Enumeration outAdjacentV(Position v) { return utNaboer(v).elements(); }`.)

3.2

Programmer først en metode

```
Queue TS(DiGraph g)
```

som tar som input en vanlig rettet graf `g` og returnerer en sekvens av dens noder i en topologisk ordning (dersom en slik finnes). Bruk kun operasjonene fra `interface DiGraph` og `interface Queue`. (Begge er spesifisert i vedlegget.) Du kan anta at du har gitt en klasse

```
class KØ implements Queue
```

samt eventuelle implementasjoner av andre `interface`'r du måtte ha bruk for. Noder V i grafer er objekter av

```
class Node implements Position {
  private int D;   public int D() { return D; }
  public void setD(int i) { D=i; }
...  }
```

3.3

Ved topologisk sortering av en multigraf, når vi fjerner en node fra multigrafen og plasserer den i output sekvensen, fjerner vi også *alle* kanter fra denne noden til noder som fortsatt er i multigrafen

– har vi to kanter fra u til v, og u fjernes, vil v stå igjen med de to kantene mindre.

1. Identifiser de stedene i algoritmen din fra 3.2 som måtte justeres for å oppnå topologisk sortering av en multigraf, dvs. når input til metoden er en `Multigraf m`.
2. Design en `interface Tilpass` som skal ivareta de variasjonene du identifiserte over. Programmer så en metode
   ```
   Queue multiTS(DiGraph g, Tilpass var)
   ```
   som skal returnere en topologisk sortering (dersom en slik finnes) av noder V i graf `g` (som muligens er en multigraf).
3. Programmer to implementasjoner av `interface Tilpass`: den ene skal anta at den aktuelle grafen `g` er en `Multigraph` mens den andre at den er en vanlig `DiGraph`. Begge skal brukes i metoden over for å lage en topologisk sorterting av `g` – avhengig av typen til `g`.

Løsningsforslag (ufullstendig men tilstrekkelig)

Opp. 1 – Fargelegging

1.1. Datastrukturen

Bruker (ikke-rettet) graf der noder (land) A og B er forbundet med en kant hvis og bare hvis de har en felles grense. `Kart` er et objekt av en klasse som implementerer `interface Graph`. Objekter som skal lagres ved grafens noder er `Land` – representasjon av kanter er uvesentlig.

```
class Land{
  private int max=4;
  private boolean[] farger= new boolean[max];
  private int farge;
  public Land() { for (int i=0;i<max;i++) farger[i]=true; }
  public void setFarge(int i) { farge=i;}
  public int farge() {
    int i=0; boolean ok= false;
    while (i<max && !ok) {
      if (farger[i]) ok=true; else i++; }
    if (ok) return i; else return max; }
  public boolean mulig() { return farge()<4; }
  public void remove(int f) { farger[f]=false; }
...}

Graph Kart;
```

1.2. Fargelegging

Algoritmen farger land med dets `farge()` og da fjerner denne fra mulige farger hos naboland.

```
boolean fargelegg() {
  Enumeration Ve= Kart.vertices(); boolean mulig=true;
  while (Ve.hasMoreElements() && mulig) {
    Land v= (Land)Ve.nextElement();
    if (v.mulig()) {
      int f= v.farge(); v.setFarge(f);
      Enumeration A= Kart.adjacentV(v);
      while (A.hasMoreElements() {
        ((Land)A.nextElement()).remove(f); }
    else mulig=false;
   }
return mulig; }
```

$\mathcal{O}(n * k)$ der $n = \#$land og $k = \#$grenser (kanter)

Det er ikke opplagt at denne algoritmen alltid klarer å fargelegge et kart – selv om det er mulig. For å sikre at en løsning alltid vil bli funnet, burde vi essensielt prøve alle muligheter. Dvs. når vi kommer til et punkt der det er umulig å farge et land, burde vi 'backtrack'e og prøve en annen farge for det landet som ble farget like før. Er ikke det mulig, burde vi gå enda et steg tilbake og prøve forrige land.

Opp. 2 – Morse

2.1. Dekoding

Bruker et binært tre der venstre barn svarer til en prik og høyre barn til en strek.

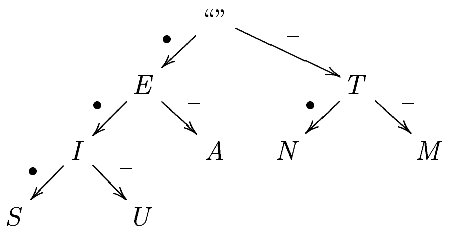
1. Min hele datastruktur er `BinTree m`. Jeg definerer hjelpemetode

```
Position of(Position p, char c) {
```

```
if (c == '.')  return m.leftChild(p);
else if (c == '-') return m.rightChild(p);
else return null; }
```

Som objekter lagret i treet bruker jeg `String` med enkle bokstaver.

2.



3.
```
String find(String ms, Position p) {
  if (p == null) return ``'';
  else if (ms.length() == 0) return p.elem();
  else return find(ms.substring(1,s.Length()), of(p, ms.charAt(0)));}
```
Selve dekoding metode blir da:
```
  String find(String ms) {
    String s= find(ms, m.root()); }
    if (s.equals(``'')) throw new NotFoundExc(); else return s; }
```

4. Algoritmen er av orden $\mathcal{O}(n)$ siden den utfører (høyst) n steg nedover treet.

5.
```
String dekode(String mo) throws NotFoundExc {
String res= ``''; int i= 0;
while (i < mo.size()) {
  String s= ``'';
  while (i < mo.size() && mo.charAt(i) != ' ')
    { s= s + mo.charAt(i); i++; }
  res = res + find(s);
  i++;}
return res; }
```
Vi går gjennom input-strengen 1 gang, dvs. vi har $\mathcal{O}(n)$ der n er lengden av denne strengen. Hver gang vi finner en kodet bokstav (mellomrom) utfører vi `find(s)` som er av orden $\mathcal{O}(|s|)$ – tilsammen vil alle disse finn-operasjonene være av orden $\mathcal{O}(n)$, som er kompleksiteten av hele algoritmen.

2.2. Koding

1. For å finne koden til en bokstav i datastrukturen fra forrige deloppgave, kan vi risikere å måtte lete gjennom hele treet som har D noder. Dette vil altså gi en $\mathcal{O}(D)$ dekoding av enkel bokstav.

2. Jeg velger heller å samle alle bokstavene i en sortert array nummerert fra 0 opptil $D-1$. Jeg har altså en enumerering av mine bokstaver og antar at konvertering (fra bokstav til dens nummer) er $\mathcal{O}(1)$. (Dette kan gjøres f.eks. ved en passende bruk av ASCII koder: gitt en bokstav, finner jeg (i $\mathcal{O}(1)$) dens ASCII-kode, og så konverterer dette tallet til et tilsv. tall mellom 0 og D.)

   Array'et mitt lagrer objekter som er strenger av prik-og/eller-strek tilsvarende Morse koder for enkelte bokstaver. Jeg har altså:
```
  String[] koder= new String[D];
```
   samt en metode
```
  int intKode(char bokstav);
```

3. For å finne koden til en gitt bokstav (gitt som `char`) utfører jeg binært søk i `koder`:
```
String kode(char bokstav) {
  return = koder[binSøk(intKode(bokstav))]; }
```
der den siste metoden er som i boken. Den er av orden $\mathcal{O}(log(D))$.

4. `String koding(String T);`
```
String res= ''; int i=0;
while (i < T.length()) {
  res= res + kode(T.charAt(i)) + ' ';
  i++;}
return res+' ';}
```
Vi utfører t ganger `kode(c)` og får da $\mathcal{O}(t * log(D))$.

Opp. 3 –

3.1. boolean erLike(Bag a, Bag b) {

```
Enumeration a1= a.elements(); boolean ok= true;
while (a1.hasMoreElements() && ok) {
  Object o= a1.nextElement();
  if (a.multipl(o) != b.multipl(o)) ok= false; }
Enumeration b1= b.elements();
while (b1.hasMoreElements() && ok) {
  Object o= b1.nextElement();
  if (a.multipl(o) != b.multipl(o)) ok= false; }
return ok; }
```

3.2. Queue TS(DiGraph G) {

```
Queue Q= new Kø(), R= new Kø();
Enumeration V= G.vertices();
while (V.hasMoreElements()) {
  Node v= (Node)V.nextElement();
  v.setD(G.inDegree(v));
  if (v.D()==0) Q.enqueue(v); }
while (!Q.isEmpty()) {
  v = (Node)Q.dequeue();
  Enumeration A= G.outIncidentE(v);
  while (A.hasMoreElements()) {
    Position e= (Position)A.nextElement();
    Node x= (Node)G.destination(e);
    x.setD(x.D()-1);
    if (x.D()==0) Q.enqueue(x); }
  R.enqueue(v); }
return R; }
```

3.3. Multigraf TS

1. Stedene merket med '**bold**' over.
2. Må generalisere disse to stedene til å håndtere evt. multigrafer:
```
interface Tilpass {
  int dec(DiGraph m, Position v, Position e);  // multiplicitet av e bland ut-kanter
fra v
  Enumeration outIncident(DiGraph m, Position v); }
```
```
Queue SeqTS(DiGraph g, Tilpass var) {
```

```
Queue Q= new Kø(), R= new Kø();
Enumeration V= g.vertices();
while (V.hasMoreElements()) {
  Node v= (Node)V.nextElement();
  v.setD(g.inDegree(v));
  if (v.D()==0) Q.enqueue(v); }
while (!Q.isEmpty()) {
  v = Q.dequeue();
  Enumeration A= var.outIncident(g, v);
  while (A.hasMoreElements()) {
    Position e= (Position)A.nextElement();
    Node x= (Node)g.destination(e);
    x.setD(x.D()-var.dec(g,v,e));
    if (x.D()==0) Q.enqueue(x); }
  R.enqueue(v); }
return R; }
```

3.

```
class Mul implements Tilpass {
  public int dec(DiGraph g, Position v, Position e) {
    Multigraph m= (Multigraph)g;
    return m.utKanter(v).multipl(e); }

  public Enumeration outIncident(DiGraph g, Position v) {
    Multigraph m= (Multigraph)g;
    return m.utKanter(v).elements(); } }

class Dig implements Tilpass {
  public int dec(DiGraph g, Position v, Position e) {
    return 1; }

  public Enumeration outIncident(DiGraph g, Position v) {
    return g.outIncidentE(v); } }
```