Unike tall

Dette var en enkelt nok forklart oppgave. Gitt en liste med tall, finn det laveste tallet som kun er i listen én gang, eller avgjør at ingen slike tall finnes.

Deloppgave 1
Svaret er garantert å være maksimalt 10, så vi kan sjekke om noen av disse tallene fungerer.

```
for (int v=1; v<=10; v++) {
  int tilfeller = 0;
  for (int i=0; i<n; i++) {
    if (tall[i] == v) tilfeller++;
  }
  if (tilfeller == 1) {
    System.out.println(v);
    return;
  }
}
//ingen av tallene fungerte
System.out.println(-1);
```

Deloppgave 2
Her kan svaret være stort, så vi kan ikke lenger sjekke alle mulige tall. Men vi kan sjekke alle tall som er i listen!

```
int svar = -1;
for (int j=0; j<n; j++) {
  int tilfeller = 0;
  for (int i=0; i<n; i++) {
    if (tall[i] == tall[j]) tilfeller++;
  }
  if (tilfeller == 1) {
    if (svar == -1 || svar > tall[j]) {
      svar = tall[j];
    }
  }
}
System.out.println(svar);
```

Deloppgave 3

Løsningen for deloppgave 2 har *kvadratisk* kjøretid. Enkelt forklart betyr det at dersom man dobler N så vil kjøretiden ganges med ca. 4. Når N kan være opp til 50 000 så blir det litt i meste laget.
En løsning er å bruke et map for å telle opp antall forekomster av hver bokstav. Eventuelt så kan man sortere listen i stigende rekkefølge. Siden tall da vil alltid være ved siden av duplikatene sine så vil svaret være det første tallet som ikke er lik noen av naboene sine.

```
Arrays.sort(tall);
for (int j=0; j<n; j++) {
  if ((j == 0 || tall[j] != tall[j-1]) &&
      (j+1 < n || tall[j] != tall[j+1])) {
    System.out.println(tall[j]);
    return;
  }
}
System.out.println(-1);
```

Warpzone

Dette er en ganske typisk graf-oppgave. Hvert brett kan betraktes som en node, og alle brett utenom det siste har 4 kanter ut - 1 til neste bane og 3 gjennom warpzonen. Oppgaven spør kun om raskeste vei fra start til slutt i en uvektet graf.

Deloppgave 1
Med N såpass liten så er denne oppgaven mulig å løse selv med Floyd-Warshall algoritmen.

```
//array for å lagre avstand fra node a til b for alle par a,b
int[][] avstand = new int[N+1][N+1];

for (int i=1;i<=N;i++)
  avstand[i][i]=0;

for (int i=1;i<=N;i++)
  for (int j=1;j<=N;j++)
    //stort tall som vi kan betrakte som uendelig for initialisering
    avstand[i][j] = 1000000000;

for (int i=1;i<N;i++) {
  avstand[i][i+1] = 1; //kan gå til neste bane
  avstand[i][warpzone[i][0]] = 1; //kan bruke første snarvei fra i
  avstand[i][warpzone[i][1]] = 1; //kan bruke andre snarvei fra i
```

```
  avstand[i][warpzone[i][2]] = 1; //kan bruke tredje snarvei fra i
}

for (int k=1;k<=N;k++) {
  for (int i=1;i<=N;i++) {
    for (int j=1;j<=N;j++) {
      if (avstand[i][j] > avstand[i][k] + avstand[k][j])
        avstand[i][j] = avstand[i][k] + avstand[k][j]
    }
  }
}
//avstand[i][j] er nå antall brett man må fullføre for å komme fra
//starten på bane i til starten på bane j for alle par i, j.
//vi må også fullføre bane N så vi legger til 1 på svaret.
System.out.println(avstand[1][N] + 1);
```

Deloppgave 2

Siden alle snarveiene går framover så kan vi løse denne deloppgaven med dynamisk programmering. La `avstand[i]` være det minste antall brett man har igjen å fullføre hvis man er på brett *i*. Denne verdien kan beregnes ved å kun se på verdiene av `avtand[j]` der *j* er høyere enn *i*. Altså

$$\text{avstand}[i] = min \left\{ \begin{array}{c} \text{avstand}[i+1]+1 \\ \text{avstand}[\text{warpzone}[i][0]]+1 \\ \text{avstand}[\text{warpzone}[i][1]]+1 \\ \text{avstand}[\text{warpzone}[i][2]]+1 \end{array} \right\}$$

Dermed kan `avstand[i]` beregnes for alle *i* ved at man tar en løkke fra N-1 ned til 1.

Deloppgave 3

Her må man bruke en mer effektiv algoritme enn i deloppgave 1. Den beste for å finne kortest sti fra ett punkt til et annet i en generell uvektet graf er bredde-først søk. Da har man en kø med alle nodene man skal utforske, som initielt kun inneholder startnoden som har avstand 0. Når man fjerner en node V fra køen så sjekker man alle kantene (V→W) ut fra den noden. Hvis noen av disse peker på en node W som enda ikke er utforsket så setter man avstand[W] = avstand[V] + 1 og legger W inn i køen. Så bare fortsetter man til man enten har funnet avstanden til sluttnoden, eller evt. til det ikke lenger er noen noder i køen.

Kuponger

Her begynte ting å bli vanskeligere. Man skulle finne ut hvor mye rabatt man kunne få - men med det problemet at dersom man får mye i rabatt enn hva noen av rabattkupongene tillater så blir kjøpet ugyldig.

Deloppgave 1
Siden alle rabattkupongene er like, så kan man bare legge til flere og flere rabattkuponger til det ikke lenger lar seg gjøre - altså til enten man er tom for kuponger eller til at neste kupong vil gjøre at sluttsummen blir under m_i verdiene.

Deloppgave 2
Siden alle rabattkupongene har like mye verdi så kan vi like gjerne bruke de "letteste" - altså de hvor m-verdiene er så lave som mulig. Sorter kupoengen i stigende rekkefølge etter m-verdi, og legg til flere og flere kuponger helt til neste kupong vil gjøre at sluttsummen blir lavere enn den kupongens m-verdi.

Deloppgave 3
Siden alle kupongene har samme m-verdi (la oss bare kalle denne for m), så er denne verdien en minste begrensning på hvor lite vi kan få prisen. Siden prisen ikke kan gå under m (med mindre N allerede er mindre enn m, slik at vi ikke trenger å bruke noen kuponger) så er det meste vi kan få i rabatt N-m. Vi må altså finne en delmengde av r-verdiene som summeres opp til et tall som er så høyt som mulig, men som ikke overskrider N-m. Dette kjent som 0-1 knapsack problemet, en klassiker innenfor dynamisk programmering.

```
bool[][] mulig = new bool[N+1][K+1];
//mulig[a][b] = true betyr det er mulig å
//oppnå sluttsum a ved å bruke de første K kupongene
mulig[N][0] = true;
for (int i=0; i<K; i++) {
  for (int v=0; v<=N; v++) {
    if (mulig[v][i]) {
      //mulig å få samme sum ved å ikke bruke denne kupongen
      mulig[v][i+1] = true;
      if (kuponger[i].r <= v) {
        //mulig å få en lavere sum ved å bruke denne kupongen
        mulig[v-kuponger[i].r][i+1] = true;
      }
    }
  }
}
for (int v=m; v<=N; v++) { //kun beløp med verdi minst m er oppnåelige
  if (mulig[K][v]) {
    System.out.println(v);
    return;
  }
}
//hvis ikke annet går, så kan vi alltid la være å bruke noen kuponger
```

```
System.out.println(N);
```

Deloppgave 4
Siden K er såpass liten her så er det mulig å løse oppgaven med *brute force*. Altså, prøv ut alle mulige kombinasjoner av kuponger, sjekk om kombinasjonen er gyldig, og spar på hva den laveste mulige sluttsummer ble.
Man kan brute force enten ved å bruke en rekursiv funksjon, eller ved å bruke et *bitmask*.

Deloppgave 5
Her er K for stor for brute force, men vi vet allerede fra deloppgave 3 at det generelle problemet er ganske likt 0-1 knapsack, bare litt vanskeligere. Fra deloppgave 2 har vi ideen om at det er lettest om vi betrakter kupongene med lav m-verdi først, og faktisk skal det ikke så veldig mye mer til enn å kombinere disse to tankene.
Dersom vi har sortert kupongene etter m-verdi før vi fyller ut `mulig`-tabellen, så holder det å endre den siste løkken til

```
int bestLøsning = N;
for (int maxKupong=0; i<m; i++) {
  for (int v=kuponger[maxKupong].m; v<=N; v++) {
  if (mulig[maxKupong+1][v] && v < bestLøsning) {
    bestLøsning = v;
  }
}
System.out.println(bestLøsning);
```

Regnskapssvinsdel

Her har vi en liste med tall som vi må bearbeide for å gjøre den gyldig. En gyldig liste er her en hvor alle prefiks-summene er ikke-negative, og den eneste måten å bearbeide listen er å slette elementer. Oppgaven ber om det færreste antall elementer som må slettes for å lage en gyldig liste.

Deloppgave 1
Listen her er såpass kort at vi kan se på alle mulige måter å slette elementer fra. Med 20 elementer så er det $2^{20} \approx 1\,000\,000$ å fjerne ingen eller flere elementer fra listen, så vi kan teste ut alle kombinasjoner, se hvilke som er gyldige, og se hvilken av de som har det færrest antall slettinger.

Deloppgave 2

Vi bygger opp en liste over de tallene vi skal ha med på følgende måte. Begynn med en tom liste, og ha lagret at summen S av alle elementene i listen er 0. Ta elementene i input-listen i rekkefølge og legg de til i listen og beregn ny verdi av S. Dersom S nå er mindre enn 0 så må vi fjerne elementer fra listen slik at S blir 0 eller større. Det vil alltid være mulig å fjerne elementet vi nettop la til, men dersom det er noen elementer i listen som er mer negative enn det vi nettop la til så vil det lønne seg å fjerne ett av de i stedet. Det optimale er å alltid fjerne det minste tallet (altså det negative tallet med størst absolutt-verdi) før vi går videre.

Behandlingen av ekspempelet i oppgaven vil da være

Input	Resultatliste	S	
-1, 5, -4, -2, 2, -5, -3, 10, -9	[ ]	0	
5, -4, -2, 2, -5, -3, 10, -9	[1]	-1	S er mindre enn 0; vi sletter det minste (eneste) tallet -1
	[ ]	0	
-4, -2, 2, -5, -3, 10, -9	[5]	5	
-2, 2, -5, -3, 10, -9	[5, -4]	1	
2, -5, -3, 10, -9	[5, -4, -2]	-1	S er mindre enn 0; vi sletter det minste tallet -4
	[5, -2]	3	
-5, -3, 10, -9	[5, -2, 2]	5	
-3, 10, -9	[5, -2, 2, -5]	0	
10, -9	[5, -2, 2, -5, -3]	-3	S er mindre enn 0; vi sletter det minste tallet -5
	[5, -2, 2, -3]	2	
-9	[5, -2, 2, -5, -3, 10]	12	
	[5, -2, 2, -3, -9]	3	

Med det resultatet at vi slettet 3 elementer.

Deloppgave 3
Denne løses likt som deloppgave 2, men vi må bare ha en mer effektiv måte å finne det laveste elementet på, slik at vi unngår å måtte traversere hele listen vår for hver gang vi skal slette et element. Dette kan gjøres ved å bruke en prioritetskø (ofte implementert som en min-heap). De fleste språk har denne datastrukturen som del av standardbiblioteket sitt (priority_queue i C++, PriorityQueue i java, eller heapq biblioteket i python), så da er det bare å være kjent med hvordan man bruker disse.
Prioritetskøer er blandt annet viktig å beherske for å kunne bruke Djikstra-algoritmen for korteste sti i grafer når kantene har forskjellige vekter.

Myntgraf

Dette var desidert den vanskeligste oppgaven, med kun 5 100-poengs løsninger. En del av deloppgavene var mulige å løse uten så mye problemer, men det var få som gjorde det også.

Deloppgave 1
Vi observerer at nøyaktig hvilke rader og kolonner vi legger mynter på ikke er så farlig. Vi kan f.eks. alltid bytte om på rader/kolonner eller fjerne rader/kolonner som det ikke ligger mynter på og få en like gyldig løsning. Siden N ≤ 5 så vet vi dermed at en løsning må få plass inn i et 5×5 rutenett. Siden det dermed bare er 25 mulige posisjoner for hver av 5 mynter så burde det gå

helt fint å prøve ut alle mulige kombinasjoner av posisjoner og se om noen av de har beskrivelsen vi er ute etter. Dersom man er litt bekymret over kjøretiden på dette så kan man også anta at den første mynten ligger i rute (0,0) og trenger dermed bare å brute-force over de restrerende 4 (eller færre) myntene.

Deloppgave 2

Siden ingen brikker er med i mer enn 2 par så betyr det at grafen må bestå av en samling med isolerte punkter, kjeder (A ↔ B ↔ C ↔ D) og sykler (f.eks. A ↔ B ↔ C ↔ D ↔ A). Vi kan analysere alle mulige konstruksjoner og ser at vi kan plassere de på rutenettet som følger

- Isolerte punkter
    - Disse bare ligger på en egen rad og kolonne
- Kjeder
    - A og B ligger på samme rad, C ligger på samme kolonne som B, D ligger på samme rad som C, etc.
- Sykler
    - Dersom sykelen har lengde 3 så ligger punktene på samme rad
    - Dersom sykelen har partallslengde så har vi samme konstruksjon som for en kjede, bortsett fra at siste brikke også skal ligge på samme kolonne som A.
    - Dersom sykelen har oddetallslengde og er lengre enn 3 så kan vi ikke veksle melom rad og kolonne, så problemet er dermed ikke løsbart.

Ved å kode opp identifisering og håndtering av alle disse tilfellene så vil man kunne løse denne deloppgaven.

Deloppgave 3

Her er tallene fortsatt ganske små, så vi kan se på andre måter å brute force problemet. La oss betrakte grafen hvor vi på en eller annen måte har laget en tilordning av hvilke kanter som er horisontale og hvilke som er vertikale. For at en tilordning av retning på kantene skal være gyldig så må vi ha at de horisontale kantene utgjør en samling med uavhengige komplette grafer (dvs. grupper med noder der alle nodene i samme gruppe har en kant mellom seg, men det er ingen kanter mellom noder i forskjellige grupper) - og tilsvarende for de vertikale kantene. Dette er ekvivalent med å si at de horisontale/vertikale kantene gir en *transitiv* relasjon - altså at om det er en horisontal kant A ↔ B og en horisontal kant B ↔ C, så må det også finnes en horisontal kant A ↔ C (og tilsvarende for vertikale kanter). Dette lar oss sjekke om en tilordning av retning på kanter er gyldig, så da kan vi bare prøve alle mulige tilordninger (det er maksimalt 2^{20} = 1 048 576 slike tilordninger) helt til vi finner en som er gyldig.

Når vi har en gyldig tilordning så må vi bare oversette denne til koordinater. Dette kan gjøres ved å ta en vilkårlig node, plassere den på rad 0, og plassere alle andre noder som har en horisontal kant til den noden også på rad 0. Finn en ny node som ikke har rad, og plasser den og alle andre noder som er forbundet til den med en horistonal kant på rad 1, osv. Når alle nodene har fått en rad kan du tilordne de kolonner ved å se på de vertikale kantene.

Deloppgave 4/5

Disse er egentlig ganske like, bare med at oppgave 4 er en del mer fleksibel i kjøretid.
Vi kan basere oss myepå hva vi har fra deloppgave 3, men bare legge til observasjonen fra deloppgave 2 om at en trekant betyr at alle nodene deler rad/kolonne, med andre ord at kantene har samme retning - mens de to kantene i en “ikke-trekant” (A ↔ B ↔ C, hvor det ikke finnes noen kant mellom A og C) må ha forskjellige retninger.
Ta en vilkårlig kant U ↔ V i grafen, og gi den en vilkårlig retning. Se på alle kantene som går ut fra U eller V. La oss f.eks. anta at V ↔ W er en kant. Dersom U ↔ W også eksisterer så vet vi at V ↔ W har samme retning som U ↔ V, mens hvis U ↔ W ikke eksisterer så vet vi at V ↔ W har forskjellig retning fra U ↔ V. Ved å søke utover i grafen kan vi gi retning på alle kanter som er i samme komponent som U og V. Når vi ikke lenger kan nå nye noder så kan vi finne en vilkårlig kant i en annen komponent og gi den en vilkårlig retning, og fortsette slik til alle kantene har fått en retning. Deretter er det bare å fullføre som i deloppgave 3.