NIO Runde 2 2014/2015 - Oppgaveløsninger

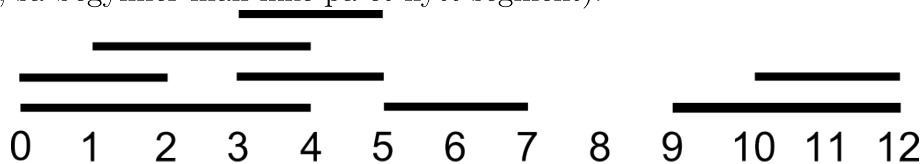
1 Stigespill

Denne oppgaven gikk ut på å simulere en runde med stigespill på et vilkårlig brett. For å holde styr på stigene kunne man bruke en array der verdien i posisjon `i` er `j` dersom det går en stige fra rute `i` til rute `j`. Når en spiller lander på en rute er det da bare å slå opp i arraet for å finne ut om det er en stige og hvilken rute sigen tar spilleren til.

Når man skulle simulere hva som skjedde måtte man holde styr på hvilken spiller sin tur det var. Dette kunne enten gjøres med en boolsk variabel som man endrer fra `true` til `false` avhengig om det er Lise eller Martin sin tur, eller man kunne se at hvis terningkastet er et partall (hvis man begynner å telle på 0) så er det Martin sin tur og ellers er det Lise sin tur.

2 Brutopia statsbaner

Her skulle man finne ut hvor mange billetter man kan klare seg med, gitt at folk går av og på toget, og at man til en hver tid må ha minst så mange billetter som det er på toget. Hvis man deler reisestrekningen opp i segmenter der det til en hver tid er minst én person om bord på toget, så ser man at hvert slikt segment krever like mange billetter som det er folk på toget på det meste - og at det er ingen måte å dele billetter mellom slike segmenter. (Hvis siste person går av toget på samme stasjon som det kommer nye på, så begynner man ikke på et nytt segment).



Her er togstrekningene folk reiser tegnet inn som svarte streker. Vi har da et segment fra stasjon 0 til 7 som krever 4 billetter (siden det er 4 personer på toget mellom stasjon 3 og 4), og ett segment fra stasjon 9 til 12 som krever 2 billetter. Totalt vil dette kreve 6 billetter.

For å lage et program som kan finne ut hvor mange billetter som denne strategien bruker så er det bare å gå gjennom alle stasjonene i rekkefølge og finne ut hvor mange personer som er på toget i det det forlater stasjonen. Når toget forlater stasjonen må vi passe på at det er minst like mange billetter om bord som det er passasjerer, og hvis toget er tomt i det det forlater en stasjon så er alle billettene vi hadde om bord tapt.

Hvis vi går gjennom alle passasjerene kan vi beregne netto endringen i antall passasjerer på hver stasjon (dvs. om 1 person går på og 3 forlater toget på en gitt stasjon så netto endringen på den stasjonen $1 - 3 = -2$. Dette kan gjøres med en array med lengde lik antall stasjoner. Dette vil desverre bli for stort for å kunne løse de siste testsettene,

ettersom det kunne være opp til $1,000,000,000$ stasjoner. Løsningen er å bruke et map og kunne holde styr på de stasjonene hvor antall personer endrer seg. (Siden det er kun $1,000,000$ passasjerer så kan det ikke være mer enn $2,000,000$ stasjoner av interesse.)

```
int __x,n;
cin>>__x>>n;
map<int, int> netto;
for(int i=0;i<n;i++) {
    int a,b;
    cin>>a>>b;
    netto[a]++;
    netto[b]--;
}

int peopleOnBoard = 0;
int ticketsOnBoard = 0;
int spentTickets = 0;

for (map<int, int>::iterator it = netto.begin(); it != netto.end(); it++) {
    peopleOnBoard += it->second;
    if (peopleOnBoard > ticketsOnBoard) {
        ticketsOnBoard = peopleOnBoard;
    }
    if (peopleOnBoard == 0) {
        spentTickets += ticketsOnBoard;
        ticketsOnBoard = 0;
    }
}
cout << spentTickets << endl;
```

3 Fjellklatring

Den første av rundens to grafoppgaver gikk ut på å finne ut hvordan man kunne forbinde noen interessante noder ved hjelp av enklet mulige stier. Det er lettest å tenke seg kartet som en graf der vi har en kant mellom alle to nabonoder (horisontalt og vertikalt) som har vekt lik vanskeligheten å bevege seg mellom de to nodene (altså absoluttverdien av høydeforskjellen). Det finnes to ganske greie måter å løse denne oppgaven på.

Den ene algoritmen er å snu oppgaven rundt. Hvis vi vet hvor dyktig en klatregruppe er så kan vi bruke bredde-først søk for å finne alle nodene de klarer å nå (vi vil da kun ha lov til å bruke kanter med vekt mindre enn eller lik ferdighetsnivået til gruppen). Hvis vi begynner i en av nodene med en attraksjon kan vi da finne ut om en gruppe med en gitt ferdighet kan besøke alle de andre attraksjonene. Ved å binærsøke på ferdighetsnivået til gruppen kan vi da finne det laveste ferdighetsnivået som lar deg besøke alle severdighetene.

Den andre algoritmen er å bruke *union-find*. `http://en.wikipedia.org/wiki/Disjoint-set_data_structure` Man sorterer da alle kantene i stigende rekkefølge etter vekt og bruker én etter én av de for å slå sammen noder i grafen. Den kanten som gjør at de siste severdighetsnodene blir slått sammen bestemmer da vanskeligheten til terrenget. Det som er litt vanskelig er å finne ut om attraksjonene kan knyttes sammen etter at en kant

har blitt brukt. Dersom man kaller attraksjonene for $A_0 ... A_k - 1$ så kan man sjekke om kantan som legges til gjør at A_0 blir forbundet med A_1. Dersom vi vet at de to er forbundet (enten fra tidligere eller fordi de ble forbundet nå) så sjekker vi heller om A_1 er forbundet med A_2, deretter A_2 med A_3, etc.

4 Flyttehjelp

Denne oppgaven gikk ut på å finne ut hvilken node som var sentret i en graf. Her er det også to ganske greie metoder for å gjøre det.

Den ene er å begynne med alle løv-noder i grafen(dvs. alle med bare én kant ut av) og fjerne alle disse. Deretter finner du alle løvnodene i rest-grafen og fjerner de. Dette gjentar du helt til du bare har igjen én node. Tegningen under viser hvordan denne algoritmen skjærer av mer og mer av grafen helt til det er kun én node igjen. For at denne algoritmen skal gå fort må man telle hvilken *grad* (dvs. hvor mange nabonoder) hver node har og justere denne graden etter hvert som man fjerner løvnoder fra grafen. De nodene som får grad 1 blir da løvnoder og skal fjernes i neste runde.

En alternativ løsning er å finne en *diameter* i grafen (dvs. to noder slik at avstanden mellom de er minst like stor som mellom noen andre noder). Noden som ligger midt i mellom disse to vil da være sentrum i grafen. For å finne en diameter i et tre kan man begynne med et bredde-først søk fra en hvilken som helst node. Den noden som ligger lengst unnna noden vi begynte på vil alltid være endepunktet i en diameter. For å finne det andre endepunktet gjør vi bare et bredde-først søk ut i fra det vi kjenner til og finner noden som er lengst unna den.

5 Ridderenes problem

Dette var dem vanskeligste oppgaven denne runden. For å løse denne måtte man benytte seg av en teknikk som heter *dynamisk programmering* som går ut på at man deler oppgaven inn i mindre deloppgaver som man kan bygge sammen.

La $m[a][k][b]$ være antall måter for ridderene fra og med 0 til og med k kle seg på, slik at ridder 0 har plagg a og ridder k har plagg b. Da har vi at $m[a][0][b] = 1$ dersom $a = b$ OG ridder 0 har rustning a, og 0 i alle andre tilfeller. Hvis $k > 0$ og ridder k skal ha på seg rustning b så må ridder $k - 1$ ha på seg en annen rustning. Altså har vi $m[a][k][b] = sum(m[a][k-1][x], x \neq b)$ dersom ridder k eier rustning b (og $m[a][k][b] = 0$ hvis ikke).

Siden ridder 0 ikke kan ha på seg det samme som den siste ridderem $(R - 1)$, så blir det totale antall kombinasjoner $sum(m[a][R-1][b], a \neq b)$. Ved å beregne alle summene underveis modulo $1,000,000,003$ så unngår man at resultatet *overflower*.