NIO Runde 2 2015/2016 - Oppgaveløsninger

1 Tannhjul

Denne oppgaven gikk ut på å se hvordan tannhjul som dreier påvirker hverandre. Dersom to tannhjul er intill hverandre og begge roterer så vil de rotere i motsatt retning av hverandre. Her ser man at dersom innholder i beholderen synker så vil alltid det første tannhjulet rotere mot klokka. Hvis det finnes et tannhjul nummer 2 så vil dette rotere med klokka, tannhjul 3 vil være mot klokka osv. Siden det siste tannhjulet må rotere med klokka når innholdet synker, så får man at måleren fungerer riktig hvis det er et partall med tannhjul.

For å finne ut om et tall er et oddetall kan man bruke *rest-ved-division*-operatoren (eller *modulo*-operatoren), som i de fleste programmeringsspråk betegnes med `%`.

```
if ((x % 2) == 1) {
  cout << "GALT" << endl;
} else {
  cout << "RIKTIG" << endl;
}
```

2 Bokstavmaskin

Her skulle man simulere en maskin som ble nevnt i første runde. Maskinen skulle bygge opp ett ord ved hjelp av 3 instruksjoner.

- `x`: Legge til en A i begynnelsen av ordet
- `x`: Legge til en B i begynnelsen av ordet
- `z`: Speilvend hele ordet

Dette kan løses enkelt, med f.eks. følgende c++ kode:

```
string ord = "A";
for (int i = 0; i < n; i++) {¨
  if (kommando[i] == 'x')
    ord = "A" + ord;
  if (kommando[i] == 'y')
    ord = ord + "B";
  if (kommando[i] == 'z')
    reverse(ord.begin(), ord.end());
}
```

(i språk hvor man ikke har en innebygd "reverse" funksjon så blir det litt mer kode.) Problemet er at ordet kunne være bygd opp av opptil 10,000,000 kommandoer. Alle disse operasjonene vil typisk ta lineær tid - dvs. at dersom ord S er dobbelt så langt som ord T, så vil det ta ca. dobbelt så lang tid å legge til bokstaver i en av endene eller å snu ordet S som det vil ta med ord T. Siden ordet vil ende opp med millioner av bokstaver så vil programmet ta allt for lang tid å kjøre. Med Stor O-notasjon `https://no.wikipedia.org/wiki/Stor_O-notasjon` sier vi at siden hver kommando bruker $\Theta(n)$ tid, så får hele algoritmen en kjøretid på $\Theta(n^2)$. For å få kjøretiden på algoritmen ned til $\Theta(n)$ så må vi få hver kommando ned til $\Theta(1)$ (eller konstant tid).

Den greieste måten å gjøre dette på er å bruke en dobbelt-endet kø `https://en.wikipedia.org/wiki/Double-ended_queue`, som lar deg legge til ting i begge ender i konstant tid. Da gjenstår det kun å få `z` operasjonen til å gå i konstant tid, men her kan man observere at man ikke nødvendigvis trenger å snu ordet hele tiden - det holder at man holder styr på om ordet er snudd eller ikke. Dersom ordet er i "normal" retning så legger man A-ene til i begynnelsen og B-ene i slutten. Dersom ordet er "snudd" så legger man A-ene til i slutten og B-ene i begynnelsen. `z`-ene vil bare bytte om tilstanden til ordet fra "normal" til "snudd" og omvendt. Til slutt gjenstår det bare skrive ut ordet i riktig rekkefølge avhengig av om det er i "snudd" eller "normal" retning.

Alternativt så kan man bruke to lister `hode` og `hale`. Til en hver tid så har vi at ordet er gitt ved `hode` i motsatt rekkefølge, og `hale` i vanlig rekkefølge. F.eks. dersom `hode` er AABA og `hale` er BBAA så vil ordet være ABAABBAA. Da er `x` å legge en `A` i slutten av `hode`, `y` å legge en `B` i slutten av `hale`, og `z` er å bytte om på hva som er `hode` og `hale`.

3 Lavalek

I denne oppgaven så skulle man finne raskeste måte å ta seg over en elv ved å hoppe mellom steiner. Dette er en variant på det velkjente finn-korteste-vei-problemet (shortest-path-problem) som er dekket på NIO bootcamp i seksjonen om grafer `http://nio.no/bootcamp/?p=117`. Løsningen er altså å gjøre et bredde-først søk, men det er noen komplikasjoner.

Dersom området man trengte å undersøke hadde vært mindre så hadde det vært fristende å lagre helekartet som et 2-dimensjonalt array. Da kunne vi bare ha "tegnet inn" alle steinene i kartet og så lett funnet ut hvilke steiner som grenser til hverandre. Desverre så kunne størrelsen på kartet være på så mye som $100,000 \times 100,000$, noe som vi hverken kan lagre eller rekke å prosessere. Denne løsningen vil derfor kun fungere på de minste datasettene.

Dersom man lager et oppslag (det som kalles et *map*) fra koordinater til steiner, så er det mulig å finne ut hvilke steiner som grenser til hverandre. Fra dette kan man lage en *naboskapsliste* (engelsk: adjacency list) representasjon av grafen, eller man kan bruke oppslaget direkte når man gjør bredde-først søk. Det blir fort litt kode som skal til for å behandle dette, men et komplett er inkludert på siste side i dette dokumentet.

4 Stemmelikhet

Dette var nok rundens vanskeligste oppgave. Her hadde man et sett med personer som alle hadde forskjellige antall stemmer, og skulle finne ut om det var mulig å ende opp med like mange stemmer for eller i mot. Her var det mange som prøvde seg med forskjellige

heurestikker, men en korrekt løsning er faktisk ganske enkel dersom man først kommer på den.

For hver sak i så fikk man alltid oppgitt én akjsonær (F_i) som fremmet forslaget og var dermed automatisk for. De restrerende aksjonærene vil enten støtte forlaget, motarbeide forslaget, eller være nøytrale. Dersom avstemningen skal bli uavgjort så må vi ha at

$$\begin{gathered}\text{[antall askjer som aksjonærene som støtter forslaget har]}\\+\\\text{[antall aksjer som } F_i \text{ har]}\\=\\\text{[antall askjer som aksjonærene som motarbeider forslaget har]}\end{gathered}$$

Hvis vi bare lager en oversikt over alle mulige antall akjser som er mulig å oppnå med akjsonærene (ekskludert F_i) så vil vi være ganske nærme en løsning. Siden det er maksimalt $100,000$ aksjer i selskapet, og maksimalt 100 aksjonærer, så er dette ganske enkelt å bygge opp med en $\Theta(N \times M)$ algoritme.

```
vector<bool> mulig(N+1,false);
mulig[0] = true; //0 aksjer er alltid mulig
for (int a = 0; a < M; a++) {
  if (a == F[i])
    continue;
  for (int j = 0; j <= N; j++) {
    if (mulig[j]) {
      //dersom vi kan oppnå j aksjer uten aksjonær a
      //så kan vi oppnå j + A[a] aksjer med aksjonær a
      mulig[j + A[a]] = true;
    }
  }
}
```

Når vi har gjort dette så gjenstår det bare å se om det finnes en verdi `t` slik at `mulig[t] == true && mulig[t + A[F[i]]] == true`. Da vil det nemmelig kunne skje at det blir uavgjort dersom F_i fremmer forslaget og får støtte med nøyaktig `t` stemmer fra andre akjsonærer.

Noe som kan virke som et problem er at vi nå også vil godta situasjoner der en person stemmer både for og mot forslaget. F.eks. dersom vi har aksjonærer med 3, 5 og 6 stemmer, så vil tabellen vår si at det er mulig å ha 8 stemmer $(3+5)$, og at det er mulig å ha 9 $(3+6)$ stemmer, men tabellen vil ikke kunne si at det er umulig å ha støtte fra 8 stemmer samtidig som man blir motarbeidet med 9. Dette blir heldigvis ikke noe problem for algorimen fordi om vi teller med aksjonæren med 3 stemmer som både å stemme for og å stemme mot forslaget, så har dette det samme resultatet som om akjsonæren hadde stemt blankt - noe som den også har lov til å gjøre.

5 Fergeforbindelse

Den siste oppgaven var igjen en graf-oppgave. Her fikk man en del øyer med forbindelser, og man skulle finne ut hvor mange nye ferjeforbindelser man trengte for å kunne forbinde alle øyene.

Dersom man skal forbinde N øyer så trenger man minst $N-1$ ferjeforbindelser. En triviell løsning for dette vil være å plukke en vilkårig øy og opprette en ferjeforbindelse fra denne til alle de andre øyene. Vanskeligheten med denne oppgaven består i at man har en del ferjeforbindelser som allerede er gitt, og det er ikke sikkert at alle disse er optimale. F.eks. hvis øyene er A, B, C og D, og det allerede finnes 3 ferjeforbindelser A-B, B-C og C-A, så vil du alikevel måtte opprette en ferjeforbindelse til slik at øy D blir koblet til nettverket.

La en *sammenhengende komponent* være et sett med øyer som er forbundet med hverandre. I eksempelet over er f.eks. A, B, C en sammenhengende komponent, og D er en sammenhengende komponent. Dersom man har K sammenhengende komponenter så vil det være mulig å forbinde alle de med $K-1$ forbindelser (f.eks. ved å forbinde en vilkårig øy i den første komponenten med en vilkårig øy fra hver av de andre komponentene). Mindre enn $K-1$ forbindelser vil ikke holde ettersom hver gang man forbinder to forskjellige komponenter så vil man bare redusere antall komponenter med 1. For å løse problemet trenger man dermed bare å telle opp antall sammenhengende komponenter og trekke fra 1.

For å finne antall komponenter kan man tenke seg følgende algoritme: Lag en liste over alle øyene. Ta privatbåt til den første øya på lista og besøk alle øyene du klarer ved å reise rundt med ferger fra denne øya. Etterhvert som du besøker øyer så markerer du de i lista di som besøkte. Når du ikke lenger kan ta ferger til nye øyer så vet du at du har fullstendig utforsket én komponent. Deretter ser du nedover på lista og finner den neste øyen du ikke enda har besøk. Ta privatbåt til den, og fortsett å utforske ved hjelp av ferger. Fortsett på denne måten til du har besøkt alle øyene. Hver gang du måtte ta privatbåt så var det fordi du begynte på en ny komponent. Siden du nå vet antall komponenter så vet du også hvor mange nye forbindelser som skal til.

Pseudo-kode for algoritmen blir dermed:

```
int komponenter = 0;
vector<bool> besøkt(N, false);
for (int i = 0; i < N; i++) {
  if (visited[i] == false) {
    komponenter = komponenter + 1;
    ...
      gjør bredde-først-søk begynnende fra node i
      marker alle nodene du kommer til som besøkte
    ...
  }
}
printf("%d\n", komponenter - 1);
```

	Tann.	Bokst.	Lava.	Stemme.	Ferge.	Totalt
Deltagere som sendte inn	51	50	38	35	38	51
Deltagere med poeng	50	47	33	32	38	50
Deltagere med full score	49	27	18	10	20	8
Innleveringer i C	2	15	68	33	4	122
Innleveringer i C++	101	436	359	443	207	1546
Innleveringer i Java	45	246	109	136	106	642
Innleveringer i Python	25	62	17	16	14	134
Totalt antall innleveringer	173	759	553	628	331	2444

lavalek.cpp

```
#include <map>
#include <queue>
#include <iostream>
using namespace std;

typedef pair<int, int> koordinat;
int main() {
  int B, L, S;
  int bi,li;
  cin >> B >> L >> S;

  //for hver stein er dette avstanden fra start-bredden til steinen
  map<koordinat, int> dist;
  //kø for bredde-først søket
  queue< pair<koordinat, int> > q;

  while (S--) {
    cin >> bi >> li;
    if (bi == B-1) {
      //steinen ligger rett ved startbredden
      dist[koordinat(bi, li)] = 1;
      q.push(make_pair(koordinat(bi, li), 1));
    } else {
      //sett til stor avstand for å markere at det finnes en stein der
      dist[koordinat(bi, li)] = 1000000;
    }
  }

  //de fire mulige hoppene man kan gjøre
  int dx[] = { 0, 1, 0,-1};
  int dy[] = {-1, 0, 1, 0};

  //selve bredde-først søket
  while (!q.empty()) {
    pair<koordinat, int> t = q.front(); q.pop();
    if (t.first.first == 0) {
      //vi er ved mål-bredden, så kun ett hopp gjenstår
      cout << t.second + 1 << endl;
      break;
    }

    //prøv de 4 mulige hoppene
    for (int i = 0; i < 4; i++) {
      koordinat nyttPunkt =
            koordinat(t.first.first + dx[i], t.first.second + dy[i]);
      if (dist.find(nyttPunkt)!=dist.end() &&
          dist[nyttPunkt]>t.second+1) {
        //det er en stein på punktet, og vi har ikke behandlet den steinen før
        dist[nyttPunkt] = t.second+1;
        q.push(make_pair(nyttPunkt, t.second+1));
      }
    }
  }
  return 0;
}
```