NIO 2018-2019 Runde 2

1 Pizzatur

Her skulle man finne ut av hvor lang tid man bruker på å besøke en rekke punkter. Siden rekkefølgen punktene skulle besøkes i var oppgitt trengte ikke programmet å gjøre noen avgjørelser av hva som er optimalt - en enkel simulering av turen fungerte fint. For hver restaurant så måtte man legge til 15 minutter til en løpende total, og dersom det ikke var det første punktet så måtte man også legge til Manhattan-avstanden til den forrige restauranten. (Manhattan-avstanden mellom to punkter er hvor langt man må gå for å komme mellom de dersom man kun kan bevege seg direkte vertikalt og horisontalt og kan beregnes ved at Manhattan-avstanden mellom (X_1, Y_1) og (X_2, Y_2) er $|X_1 - X_2| + |Y_1 - Y_2|$)

Dersom man fikk feil svar på den siste deloppgaven så var det sanynsligvis fordi man brukte 32-bits tall for å holde på svaret. Når X_i og Y_i kunne være opp til 10^6 og det kunne være 10^5 punkter så ser vi med litt overslagsregning at den totale reiseavstanden kan være opp til nesten 2×10^11. Et 32-bits tall med fortegn (`int` i C++ eller Java) kan kun inneholde tall opp til ca. 2×10^9. Man må derfor bruke 64-bits tall (`long long` i C++ eller `long` i Java).

2 Datamaskiner

Her hadde man en masse teknologier med avhengigheter i mellom seg, og man måtte finne ut av hvor mange man måtte utvikle for å finne ut av en spesifik en.

Man ser at teknologiene og teknologi-avhengighetene definerer det som kalles en *rettet graf* hvor teknologiene er noder og avhengighetene er kanter. Grafen er rettet (dvs. at alle kantenen har en retning) fordi avhengighetene bare går én vei - man trenger matematikk for å utvikle katapulter men man trenger ikke katapulter for å utvikle matematikk. Mer spesifikt så sier begrensningen om at alle teknologiene vil la seg utforske at grafen er en *asyklisk rettet graf* fordi ingen teknologier kan avhenge av noe som krever at teknologien allerede er utviklet.

Konseptuelt virker det veldig greit å lage en algoritme for å finne ut av svaret:

- Prøv å utvikle `datamaskiner`
- Dersom du på noe tidspunkt skal utvikle en teknologi som har avhengigheter, utvikl de avhengighetene først.

- Tell opp hvor mange teknologier som har blitt utviklet og skriv ut det tallet

Det er derimot ikke helt trivielt å få dette til å kjøre raskt nok til å ta alle deloppgavene. En første tilnerming vil kanskje være følgende C++ kode

```
map<string, vector<string> > avhengigheter;
set<string> utviklet;

void utvikl(string teknologi) {
  for (int i = 0; i < avhengigheter[teknologi].size(); i++)
    utvikl(avhengigheter[teknologi][i]);
  utviklet.insert(teknologi);
}

int main() {
  ... Initier avhengigheter og utviklet fra input ...
  utvikl("datamaskiner");
  cout << utviklet.size() << endl;
}
```

Men et problem her er at man kan komme til å kalle `utvikl` flere ganger for samme teknologi, og dermed behandle hver avhengighet flere ganger, noe som kan resultere til et eksponensielt antall kall til `utvikl`. Dersom man legger inn en sjekk for dette i `utvikl` metoden så kan man ungå at den blir kalt flere enn maksimalt `M` ganger.

```
void utvikl(string teknologi) {
  if (utviklet.find(teknologi) != utviklet.end()) return;
  for (int i = 0; i < avhengigheter[teknologi].size(); i++)
    utvikl(avhengigheter[teknologi][i]);
  utviklet.insert(teknologi);
}
```

3 Suluklak

Her skulle man behandle en graf som til å begynne med bestod av N noder og ingen kanter. Deretter fikk man en rekke kanter som man skulle avvise dersom de ville gjøre at node 0 og 1 blir forbundet, eller legge til i grafen dersom de ikke gjør det.

For den første kanten er det veldig enkelt å svare på spørsmålet, men det blir vanskeligere etterhvert som grafen oppdaterer seg. Den mest direkte løsningen er nok

- Legg den nye kanten til i grafen
- Gjør et bredde-først søk fra node 0 og se om du kan nå node 1
- Hvis du kan - fjern kanten fra grafen og skriv ut "nei"

- Hvis du ikke kan - behold kanten og skriv ut "ja"

Men et bredde-først søk kan ta lang tid dersom grafen er stor. Denne algoritmen vil ikke kunne løse de største datasettene.

En litt lurere algoritme vil være å se at hvilke kanter som til en hver tid er med i grafen er ikke så farlig. Det eneste som er av betydning er om noder befinner seg i samme sammenhengende komponent eller ikke. For å se om man skal avvise veien mellom `a` og `b` så kan man da sjekke om

> (komponenten med a er komponenten med 0 OG
> komponenten med b er komponenten med 1)
> ELLER
> (komponenten med a er komponenten med 1 OG
> komponenten med b er komponenten med 0).

Med noen hjemmesnekra datastrukturer for å holde styr på komponentene kan man få kjøretiden ned til $O(N^2 + M)$, men det finnes en veldig effektiv datastruktur for slike *disjunkte sett* som gir enda raskere kjøretid og som dermed kan løse den siste testsettgruppen. Hvert element har en *overordnet* som kan være seg elementet selv. Representanten til et element er elementet selv dersom den er sin egen overordnede - hvis ikke er den den samme som representanten til sin overordnede. To elementer tilhører samme sett hvis og bare hvis de har samme representant. Man kan dermed raskt slå sammen to sett vet å sette den overordnede til det ene settet sin representant til å være representanten for det andre settet. Ved å "komprimere" stiene som fører til representantene nå man spør om representantene så sørger man for at kjøretiden ikke blir lang dersom man gjør mange spørringer på en graf med veldig lange kjeder.

```
vector<int> overordnet; //initiert til 0, 1, 2, .... N-1

int representant(int e) {
  if (overordnet[e] == e) {
    return e;
  } else {
    overornet[e] = representant(overordnet[e]);
    return overordnet[e];
  }
}

void knyttSammen(int a, int b) {
  overordnet[representant(a)] = representant(b);
}

bool skalBygge(int a, int b) {
  if (representant(a) == representant(0) &&
      representant(b) == representant(1))
```

```
    return false;
  if (representant(a) == representant(1) &&
      representant(b) == representant(0))
    return false;
  return true;
}
```

4 Bare rør

Dette er et typisk optimaliseringsproblem hvor det finnes mange måter å gjøre en ting på, men du skal klare å finne den beste.

For små datasett så er det mulig å gjøre brute force forsøk. For å løse et nettverk kan du gå gjennom alle mulige par av naborør, prøve å koble de sammen, og se hva kostnaden blir av å koble sammen det restrerende nettverket. Dvs. noe slik som

```
int optimal(vector<int> a) {
  if (a.size() == 1) return 0; //har vi ett rør er vi ferdige.
  int best;
  for (int i = 0; i < a.size() - 1; i++) {
    //kobl sammen rør i med rør i+1
    vector<int> rest;
    for (int j=0;j<a.size();j++) {
      if (j < i || j > i+1) rest.push_back(a[j]);
      if (j == i) rest.push_back(a[j] + a[j+1]);
    }
    int kostnad = optimal(rest) + ((a[i]+a[i+1])*311)\%104729;
    if (i==0 || kostnad < best) best = kostnad;
  }
  return best;
}
```

Dette blir nok alt for trengt for å løse noe annet enn den første subtasken.

For å få til en løsning som gir full poengsum kan man begynne med å se på hva den siste sveisingen blir. Uansett hvor man gjøre den siste skjøten, så koster den det samme. Kostnaden av å bygge opp de to delene kan derimot være forskjellig avhengig av hvor skjøten en. Hvis den siste skjøten er mellom komponent X og $X+1$ så ser vi at vi først må ha klart å sveise sammen alle komponentene fra 0 til og med X og alle komponentene fra $X+1$ til $N-1$. Hvis vi lar $F(A,B)$ være kostnaden av å sveis sammen alle komponentene fra og med A til og med B så ser vi dermed at

$F(A,B) = 0$ dersom $A = B$

$F(A,B) =$minimum over alle X med $A \leq X < B$ av

$$F(A,X) + F(X+1,B) + (\text{[lengen av hele stykket]} \times 311)\%104729 \text{ ellers}$$

Programmerer man dette som en ren rekursjon så vil den fortsatt bruke alt for lang tid, men det er kun fordi den ender opp med å gjøre gjentatte kallene til F med de

samme verdiene for A og B. Hvis vi lagrer unna resultater vi har funnet ut av i en 2-dimensjonal array så slipper vi å beregne de om igjen. Vi ender dermed opp med en kjøretidskompleksitet på $O(N^3)$ som kjører fint innenfor inputbegrensningene.

5 Fantasy

Dette var enda et optimaliseringsproblem hvor man skulle klare å maksimere antall poeng man kunne få over et sett med runder. Den maksimale poengsummen det er mulig å få i en enkel runde er lett å finne ut av - bare summer opp de K største verdiene i den runden. Problemet er at valg av et lag i en runde setter begrensninger for hvilket lag man kan ha i neste runde. Man kan tenkte seg at alle de forskjellige lagene er noder i en graf, med kanter mellom lag som har maksimalt én spiller byttet ut mellom seg. For hver runde så får man poeng for den noden man er på, og kan deretter flytte seg til en nabonode.

De tre første deloppgavene var veldig enkle her. I den første så hadde man ikke noe valg, og måtte dermed ha med alle spillerene. I den andre så var det bare én spiller som skulle utelates i hver runde, noe som betyr at du står fritt til å sette sammen hvilket som helst lag for hver runde og kan dermed alltid ta det beste laget. I den tredje så var det bare én runde, så her var det også bare å plukke ut det beste laget.

For å løse de to siste deloppgavene så måtte man gjøre litt smartere ting for å behandle de forskjellige mulige lagene. En veldig effektiv måten å representere et lag på er med en "bitmask", der bit nummer i telt fra høyre er satt dersom spiller i er med på laget. En bitmask som `001101` vil da bety at spiller 0, 2 og 3 er med på laget. For fjerde deloppgave så kan man se for seg alle mulige lag i første runde, og for hver av disse se på hvilke mulige lag man kan bruke i andre runde.

```
int N, M;
vector<vector<int> > scores;
int antallBits(int a) {
  int s = 0;
  for (int i=0;i<N;i++) {
    if ((a & (1<<i)) s++;
  }
  return s;
}

int score(int runde, int spillermaske) {
  int sum = 0;
  for (int i = 0; i < N; i++) {
    if (spillermaske & (1<<i)) {
      sum += scores[runde][i];
    }
  }
```

```
  return sum;
}

int main() {
  ... les inn input ...
  int best = 0;
  for (int m = 0; m < (1<<N); m++) {
    if (antallBits(m) == M) {
      //m har M bits satt, og er dermed et lovlig lag
      //å ha i første runde
      int score = score(0, m);
      int bestRunde2 = score(1, m); //prøv med samme lag i andre runde
      for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
          if ((m & (1<<i)) && !(m & (1<<j))) {
            //spiller i var med på laget, og spiller j var
            //ikke med, så vi prøver å bytte de om
            bestRunde2 = max(bestRunde2, score(1, m - (1<<i) + (1<<j));
          }
        }
      }
      best = max(best, score + bestRunde2);
    }
    cout << best << endl;
  }
}
```

For siste deloppgave må man bruke *dynamisk programmering*. La $B[r][m]$ være høyeste poengsum det er mulig å ha i runde r dersom man har lag m i den runden. Denne verdien kan vi finne ut av ved å se på alle mulige naboer av lag m og se hvor mye poeng de fikk i runde $r-1$. Ved å jobbe seg opp fra runde 0 til $R-1$ kan man dermed finne ut hva den beste poengsummen det er mulig å ende opp med er.

6 Implementasjoner

```
//Pizzatur
#include <iostream>

using namespace std;

int main() {
    int n;
    cin>>n;
    long long d = 0;
    int x0,y0;
    for (int i=0;i<n;i++) {
        int x,y;
        cin>>x>>y;
        if (i != 0) {
            d += abs(x-x0)+abs(y-y0);
        }
        x0 = x;
        y0 = y;
        d += 15;
    }
    cout << d << endl;
}
```

```
//Datamaskiner
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <string>

using namespace std;


map<string, vector<string>> G;
set<string> V;

int dfs(const string &u) {
    if (V.find(u) != V.end()) return 0;
    V.insert(u);

    int res = 1;
    for (auto &v : G[u]) res += dfs(v);

    return res;
}

int main() {
int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        string u;
        int a;
        cin >> u >> a;
        for (int j = 0; j < a; j++) {
            string v;
            cin >> v;
            G[u].push_back(v);
        }
    }

    cout << dfs("datamaskiner") << endl;
}
```

```
//Suluklak
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

constexpr int MAXN = 100000;

int n, m, p[MAXN], s[MAXN];

int find(int x) {
    return x == p[x] ? x : p[x] = find(p[x]);
}

void unite(int a, int b) {
    a = find(a);
    b = find(b);

    if (s[a] > s[b]) {
        s[a] += s[b];
        p[b] = a;
    }
    else {
        s[b] += s[a];
        p[a] = b;
    }
}

int main() {
    cin >> n >> m;

    for (int i = 0; i < n; i++) {
        p[i] = i;
        s[i] = 1;
    }

    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;

        if (((find(0) != find(a)) || (find(1) != find(b))) &&
            ((find(0) != find(b)) || (find(1) != find(a)))) {
```

```
            unite(a, b);
            cout << "ja" << endl;
        }
        else {
            cout << "nei" << endl;
        }
    }
}
```

```
//Bare rør
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

constexpr int MAXN = 802;

int n;
long long L[MAXN], M[MAXN][MAXN];
bool V[MAXN][MAXN];

inline long long cost(int a, int b) {
    return ((L[b] - L[a-1]) * 31111) \% 1047291l;
}

long long dp(int a, int b) {
    if (a == b) return 0;
    if (V[a][b]) return M[a][b];
    V[a][b] = true;

    long long res = dp(a, a) + dp(a+1, b);
    for (int i = a; i < b; i++)
        res = min(res, dp(a, i) + dp(i+1, b));

    res += cost(a, b);

    return M[a][b] = res;
}

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> L[i];
        L[i] += L[i-1];
    }

    cout << dp(1, n) << endl;
}
```

```
//Fantasy football
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int bitsSet(int m, int n) {
    int s = 0;
    for (int i=0;i<n;i++) {
        if (m&(1<<i)) s++;
    }
    return s;
}
int roundScore(vector<int> &scores, int team) {
    int s = 0;
    for (int i=0;i<scores.size();i++) {
        if (team & (1<<i)) s += scores[i];
    }
    return s;
}
int main() {
    int n,k,r;
    cin>>n>>k>>r;

    vector<vector<int> > scores(r, vector<int>(n, 0));
    vector<int> teams;

    for (int i=0;i<r;i++) for (int j=0;j<n;j++) cin >> scores[i][j];

    vector<vector<int> > dp(r, vector<int>(1<<n));
    for (int m=0;m<(1<<n);m++) {
        if (bitsSet(m, n) == k) {
            teams.push_back(m);
            dp[0][m] = roundScore(scores[0], m);
        }
    }

    for (int rr=1;rr<r;rr++) {
        for (auto &m : teams) {
            int rp = roundScore(scores[rr], m);
            int bestFrom = dp[rr-1][m];
            for (int i=0;i<n;i++) {
                for (int j=0;j<n;j++) {
```

```
                    if (((m & (1<<i))) && !(m & (1 << j))) {
                        bestFrom = max(bestFrom, dp[rr-1][m - (1<<i) + (1<<j)]);
                    }
                }
            }
            dp[rr][m] = rp + bestFrom;
        }
    }

    cout << *max_element(dp[r-1].begin(), dp[r-1].end()) << endl;
}
```