Tema i oppgavene

Dette er de forskjellige temaene som er tiltenkt til hver oppgave. Merk at man kan bruke mer avanserte teknikker for å løse oppgavene, men man kommer lang vei med en god forståelse av det som er listet her.

Sokkeskuff: Sortering, grådig søk

Regnbuelakris: Grådig søk

Manngard: Dynamisk Programmering

Trebygger: Sortering, Set

Mine fienders fiender: Strongly connected components, Bitset

Sokkeskuff

I **Sokkeskuff** skal vi lage så mange sokkepar som mulig slik at hvert par består av relativt like sokker. Relativt like vil si at forskjellen i mellom nyansene er mindre eller lik T.

Subtask 1 Vi vet at alle sokker kan pares med hverandre, så vi må bare finne største mulige antall par. Det er naturligvis $\lfloor N/2 \rfloor$, hvor $\lfloor x \rfloor$ betyr x rundet ned. Dvs. $\lfloor 85.5 \rfloor = 85$.

Subtask 2 Siden det bare er to sokker trenger vi bare å finne ut om de kan matches eller ikke. Spørsmålet er da om differansen er $\leq T$.

Subtask 3 Her har vi at $T = 0$ slik at vi må ha nøyaktig like sokker for å kunne parre dem. Dette gjør at man trenger bare å konsentrere seg om hver nyanse for seg, siden de ikke kan påvirke noen andre nyanser. For hver nyanse har man et antall av den nyansen, og antall mulige par med nyansen kan løses likt som i subtaks 1. Svaret blir da

$$\sum_{k_i} \lfloor |k_i|/2 \rfloor \tag{1}$$

hvor hver k_i er fargenyansene og $|k_i|$ er antallet sokker av nyansen k_i. Merk at man ikke kan summere opp alle nyansene til K, men man må bare se på de som vi faktisk har sokker av.

Full løsning Vi starter med å se på den sokken med den laveste fargenyansen. Vi kan potensielt parre den med mange forskjellige sokker, men det vil alltid lønne seg å parre den med den nest laveste sokken. Et resonnement kan være som følger:

La a være fargen til den laveste sokken, og vi vurderer å parre den med b og c, hvor $b < c$. Hvis vi matcher a og b, vil vi ha c igjen som kan pares med sokker opp til $c + T$. Hvis tilsvarende blir det hvis vi matcher a og c: at vi har b igjen som kan parres opp til $b + T$. Siden vi antok at a er lavest, vil det ikke være noen sokk som b kan parres med, men som ikke c kan parres med. c når uansett a, og $b + T < c + T$. Det er derfor c som gir oss best sjanse videre (kan brukes med potensielt høyere verdier), og vi vil derfor spare på den og heller bruke b.

En fremgangsmåte for oppgaven kan da være å sortere listen i stigende rekkefølge. Vi går da gjennom listen og hvis vi ikke har paret sokk i vil vi prøve å parre den med sokk $i + 1$. Hvis det ikke går får vi ikke parret sokk i med noen, så vi går videre, og teller fullførte par.

Listing 1: C++-løsning av Sokkeskuff

```cpp
#include<bits/stdc++.h>
using namespace std;

int main()
{
    int n, k, t;
    cin >> n >> k >> t;

    vector<int> sokker(n);
    for (int &i : sokker) cin >> i;

    sort(sokker.begin(), sokker.end());

    int output = 0;
    for (int ind = 0; ind < sokker.size() - 1; ind++)
    {
        if (sokker[ind + 1] - sokker[ind] <= t)
        {
            output++;
            ind++;
        }
    }

    cout << output << "\n";
}
```

Regnbuelakris

I **Regnbuelakris** skal vi prøve å dele opp en lakris stang i så mange biter av lengde K med K forskjellige smaker.

Subtask 1 Her er det bare to forskjellige farger, og hver bit trenger de to fargene. Siden vi ikke kan dele en cm opp til å brukes i flere biter, er det ikke helt nok å bare telle antallet ganger stangen skifter farge. Det må også ha vært to like på rad (unntatt start og slutt). Med andre ord vil vi telle antallet ganger hvor stangen skifter farge og forrige bit ikke allerede har blitt brukt til en annen bit.

Subtask 2 Siden $N \leq 10000$ som for lineære søk regnes som et lite tall, kan vi for hver indeks i starte og se om de neste K centimeterene er av forskjellige farger. Hvis de K bitene er forskjellige har man en gyldig bit, og man hopper til indeks $i + K$, siden man har brukt de foregående. Det vil alltid være optimalt å ta de tidligste bitene man kan finne ettersom det gir flest mulig centimeter igjen som man kan bruke senere til å finne flere biter. Løsningen blir da i $O(NK)$ tidskompleksitet.

Subtask 3 Her har vi at det bare er smak 1 som kommer flere ganger. Da vet vi at hvis et stykke av lengde K ikke er gyldig betyr det at det er to 1-ere i biten. Det gir at for at en bit skal være gyldig må det være K centimeter siden slutten av forrige bit og det kan maks være en 1-er blant de siste K smakene. For å sjekke dette kan man f.eks. huske på de to siste 1-erene, eller ha en prefix sum for å finne 1-ere i et intervall.

Subtask 4 Denne subtasken er i stor grad samme som full løsning.

Full løsning Her funker ikke løsningen vi gjorde i subtask 2, ettersom $O(NK)$ blir for tregt når $N \leq 200000$ og $K \leq 100000$. Vi må derfor finne en mer effektiv måte å holde styr på om vi har duplikater i intervallet vårt. Det er flere måter å gjøre dette på og her er en av dem:

Vi ønsker at intervallet vi ser på aldri skal ha to like biter. Vi kommer til å utvide intervallet ved å legge til en ny centimeter, men hvis den allerede finnes fjerner vi starten av intervallet helt fram til vi har fjernet den gamle versjonen av den nye smaken. På denne måten kan vi sjekke om lengden av intervallet er K, og hvis ja, starter vi neste intervall ved neste node. For å implementere dette kan man bruke en boolean liste for å holde styr på om en smak allerede finnes i intervallet, og en kø for å representere rekkefølgen. Hvis du skal legge til en smak som allerede finnes, fjerner du fra starten helt til du fant den samme smaken du skal legge til. Den legger du til på slutten av køen. Hvis køen har lengde K har du en gyldig bit, og du tømmer både listen og køen din, og fortsetter videre. Kan gjøres i $O(N)$ tid.

Listing 2: C++-løsning av Regnbuelakris som bruker set

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n, m, k;
    cin >> n >> m >> k;

    set<int> seen;
    int output = 0;

    vector<int> v(n);
    for (int &i : v) cin >> i;
    int erase_ind = 0;

    for (int i = 0; i < n; i++)
    {
        while (seen.count(v[i]))
        {
            seen.erase(v[erase_ind++]);
        }
        seen.insert(v[i]);

        if ((int)seen.size() == k) {
            output++;
            seen.clear();
            erase_ind = i + 1;
        }
    }

    cout << output << "\n";
}
```

Manngard

I **Manngard** får vi N rekker hvor hver rekke har 2 personer. De skal bære fakler på en sånn måte at hver person er lyst opp av minimum en fakkel. Hver fakkel dekker personen som bærer den og folkene direkte ved siden av (ikke diagonalt). Man skal velge hvem som bærer fakler på en sånn måte at minst mulig krefter blir brukt.

Subtask 1 Her er det bare én rekke, så man trenger bare én fakkel og den sterkeste personen burde bære den.

Subtask 2 Nå er alle like sterke, så det eneste som har noe å si er antall fakler vi bruker. Vi må derfor finne det billigste mønsteret som dekker alle folkene. Da er det en person hver andre rad og på alternerende side:

#.

..

.#

..

#.

Det må også være en fakkel på siste og første rekke, og så blir det dette mønsteret i mellom. Dette gir $\lfloor N/2 \rfloor + 1$ fakler totalt.

Subtask 3 Her er det to rekker, slik at man trenger to fakler. Uansett hvordan man plasserer de to faklene dekker man begge radene, slik at man velger at de to sterkeste bærer.

Subtask 4 Siden $N = 10$ trenger vi ikke ha en veldig god tidskompleksitet. Her kan vi prøve ut alle mulighetene og velge den billigste. Det er totalt 20 personer og hver person kan enten bære en fakkel eller ikke bære en fakkel. Det gir $2^{20} \approx 10^6$ muligheter, og man kan prøve alle sammen.

Full løsning For full løsning må vi bruke dynamisk programmering. For å gjøre dette må vi først beskrive en DP state. For hver person har vi tre muligheter: personen er ikke opplyst foreløpig av noen andre, personen er opplyst og personen bærer en fakkel og kan dermed opplyse andre. Hvis vi da kaller en rad en dp state, har vi da 9 forskjellige muligheter som korresponderer til 3 muligheter for første person og 3 muligheter for andre person. Merk at det er 2 av de 9 som ikke er mulig, f.eks. at ene personen bærer fakkel og andre er ikke opplyst.

Med andre ord vil vi for rad i holde styr på minste styrke vi trenger for at alle radene før i skal være opplyst og at den siste raden vi har sett på skal være i hver

av de aktuelle tilstandene. Når vi har disse kan vi renge ut neste rad bare ved å bruke siste rad, ettersom valgene vi gjorde før det har ikke lenger noe å si. Det er en del tilfeller å holde styr på, så bare å holde tungen rett i munnen. For et konkret eksempel på hvordan man kan gjøre dette går det ann å se på kodefilene.

Listing 3: C++-løsning av Manngard

```cpp
#include<bits/stdc++.h>
using namespace std;
using arr = array<int, 2>;
constexpr int maxn = 5e3 + 5;

// mørk, opplyst, fakkel
int dp[maxn][3][3] = { 0 };

int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    cin >> n;

    vector<arr> v(n + 1);
    int a, b;
    // Merk at her brukes 1 indeksering
    // I noen tilfeller kan det være enklere for DP oppgaver
    for (int i = 1; i <= n; i++)
    {
        cin >> a >> b;
        v[i] = {1000 - a, 1000 - b};
    }

    for (int i = 0; i < 3; i++) for (int y = 0; y < 3; y++) dp[0][
       i][y] = 1e9;
    dp[0][1][1] = 0;

    for (int i = 1; i <= n; i++)
    {
        dp[i][0][0] = dp[i - 1][1][1];
        dp[i][1][0] = dp[i - 1][2][1];
        dp[i][0][1] = dp[i - 1][1][2];
        dp[i][1][1] = dp[i - 1][2][2];

        dp[i][2][1] = min({
            dp[i - 1][0][1],
            dp[i - 1][1][1],
            dp[i - 1][1][2],
            dp[i - 1][2][1],
```

```
            dp[i - 1][2][2]
        }) + v[i][0];

        dp[i][1][2] = min({
            dp[i - 1][1][0],
            dp[i - 1][1][1],
            dp[i - 1][2][1],
            dp[i - 1][1][2],
            dp[i - 1][2][2]
        }) + v[i][1];

        dp[i][2][2] = min({
            dp[i - 1][0][0],
            dp[i - 1][0][1],
            dp[i - 1][1][0],
            dp[i - 1][1][1],
            dp[i - 1][1][2],
            dp[i - 1][2][1],
            dp[i - 1][2][2]
        })
        + v[i][0]
        + v[i][1];
    }

    cout << min({
        dp[n][1][1],
        dp[n][1][2],
        dp[n][2][1],
        dp[n][2][2]
    }) << "\n";
}
```

Trebygger

I trebygger får vi en rekkefølge med tall og skal gi hvor tallet ender opp hvis vi følger mønsteret til et binært søketre.

Subtask 1 I denne subtasken vet vi at tallene kommer i strengt stigende rekkefølge. Det betyr at når vi skal plassere A_i så vil den bli plassert lengst mulig ned i treet på høyre side, siden den er større enn alle tidligere tall. A_{i+1} vil så bli plassert et hakk under A_i siden de begge følger samme sti ned i treet (de er begge større enn alt før), og som høyre barn siden $A_{i+1} > A_i$. Vi vet dermed at dybden til A_{i+1} vil være en større en dybden til A_i og dette gjelder for alle i. Med andre ord $0, 1, 2, \cdots N-1$.

Subtask 2 Når $N \leq 1000$ betyr det at vi kan lage en algoritme i $O(N^2)$ tidskompleksitet. En måte å gjøre det er at man for hvert nye tall (som er $O(N)$) søker man gjennom hele søke treet (som også er $O(N)$). Grunnen til at å søke i søketreet er $O(N)$ er fordi den dypeste noden kan f.eks. være $N/2$ som er i $O(N)$. Med andre ord handler subtasken om å implementere søketreet slik det er beskrevet i oppgaven, og det er raskt nok når $N \leq 1000$.

Full løsning Nå er N høy så kan vi ikke lenger ha $O(N^2)$ kjøretid. Vi må fortsatt prosessere $O(N)$ nye tall, så den biten er lik, men kan vi finne ut hvor de skal plasseres uten å søke gjennom alle tallene?

Svaret er ja. Si vi fikk sekvensen $5, 12, 8, \cdots$. Vi starter med å plassere 5 som roten, og der er det ikke noen andre muligheter. Når vi har plassert 5 deler vi opp tallene i to intervaller; de som er større enn 5 og de som er mindre enn 5. Hvert intervall vil ha sin plass, uavhengig av hvilket tall i intervallet vi velger. Når vi så har plassert 12. Deler vi så det intervallet med tall større enn 5 inn i to forskjellige intervaller igjen: større enn 5 og mindre enn 12, og større enn 12. Dette fortsetter videre med 8 osv. Vi kan skrive utviklingen som

$$(-\infty, \infty) \to (-\infty, 5)(5, \infty) \to (-\infty, 5)(5, 12)(12, \infty) \to (-\infty, 5)(5, 8)(8, 12)(12, \infty)$$

Merk at her vil hvert intervall korrespondere til en bestemt ny plassering i treet, og dermed også dybden. Hvis vi så lagrer f.eks. hvor hvert sett starter og dybden, kan vi binærsøke over start posisjoner og så erstatte det riktige intervallet med to nye. Dette lar oss finne dybden i $O(\log(N))$ tid som totalt gir $O(N \log(N))$.

Listing 4: C++-løsning av Trebygger

```
#include<bits/stdc++.h>
```

```cpp
using namespace std;
using arr = array<int, 2>;

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int n;
    cin >> n;

    vector<int> v(n);
    for (int& i : v) cin >> i;

    set<arr> intervals;
    intervals.insert({n + 1, 0});

    for (int i = 0; i < n; i++)
    {
        auto it = intervals.lower_bound({v[i], -1});
        cout << it->at(1) << "\n";

        intervals.insert({it->at(0), it->at(1) + 1});
        intervals.insert({v[i], it->at(1) + 1});
        intervals.erase(it);
    }
}
```

Mine fienders fiender

I **Mine fienders fiender** skal man svare på spørsmål om hvorvidt en person liker, misliker eller ikke har noe forhold til en annen person.

Subtask 1 Her er N, M og Q relativt små. Det gjør at vi har tid til å kjøre en brute force algoritme i $O(Q(N+M))$. Da tar vi for hver enkelt query og går gjennom grafen. Siden det ikke er noe vits å besøke samme node flere ganger eller se på hver kant flere ganger, vil hver gjennomgang ta $O(N+M)$ tid med feks. DFS.

Liten kommentar angående litt sjeldnere begrensninger Vanligvis antyder begrensingene (feks. $N \leq 50000$) hvilken tidskompleksitet vi har lyst til å oppnå. I denne oppgaven ser vi at det ligger mellom $N \approx 200000$ som ofte tyder på løsning i $O(N \log N)$ og $O(N)$, og $N \approx 1000$ som ofte tyder på løsning i $O(N^2)$. Når vi ligger i mellom betyr det gjerne at man har litt ekstra tid og at man kanskje kan prøve å benytte seg av noen triks, slik som vi skal gjøre her.

Bitsets er en datastruktur som likner på en array med heltall, men man bruker det som en boolean array, altså man ser på 0ere og 1ere i stedet for hele tall. Hver bit i et feks. 32 bits heltall koder for en boolsk verdi, slik at et 32 bits heltall kan bli sett på en boolean array med lengde 32. Fordelen med dette er at CPU-ene er laget slik at de kan gjøre forskjellige operasjoner på heltall omtrent like raskt som de gjør operasjoner på en enkelt boolsk verdi. Man kan da på den måten få veldig god konstant faktor ved at man bruker de rette CPU-instruksjonene, og dette legger bitsets til rette for. Man kan da gjerne bruke algoritmer med tregere tidskompleksitet siden den gode konstantfaktoren gjør opp for det. Ofte later man som om lineære operasjoner på bitsets har tidskompleksitet $O(N/32)$ eller $O(N/64)$ avhengig av prosessoren den blir kjørt på.

Subtask 2 I denne subtasken vil hver person enten like en annen eller ikke ha noen mening. Spørsmålene kan da stilles som: er B et barn av A? Å undersøke dette for hvert spørsmål ville vært for tregt. I stedet skal vi bruke Strongly Connected components (SCC) og bitsets. Hvis vi først later som at grafen er asyklisk kan vi bruke bitset til å holde styr på hvem som er barna til en node. Vi begynner nederst i grafen (altså ved den noden som ikke har noe barn), og går gradvis oppover (til foreldre som ikke lenger har noen barn som ikke er besøkt). Når vi kommer til en node har vi allerede prosessert alle barna, og da sier vi at denne nodens bitset er unionen av alle barnas bitset. Den prosessen vil potensielt ta $O(NM)$ tid (siden for hver kant av M kopierer vi et bitset av lengde N), men siden vi bruker bitssets er det likevel innenfor. Når vi skal besvare spørsmål kan vi slå opp i bitsettet til A om B er markert, etter at vi har gjort preprosesseringen.

Så langt antok vi at grafen var asyklisk, noe som ikke stemmer for subtasken. Vi kan immidlertid se at hvis vi har en sykel kan vi behandle hele sykelen som en node, og at hvis vi lager den kondenserte grafen hvor alle sykler blir til enkelt noder vil så grafen være asyklisk. Måten man typisk gjør dette er ved at man finner strongly connected components som er disse syklene. En strongly connected component er slik at hver enkelt node i komponenten kan direkte eller indirekte nå hver andre node i komponenten. I vårt tilfelle betyr det at hver node liker hver andre node i komponenten, og at alle par som gjennsidig liker hverandre vil være i samme komponent.

Subtask 3 Nå introduserer vi også at folk kan mislike andre folk, men vi gjør grafen asyklisk. Da trenger man ikke Strongly Connected Components. Trikset i denne subtasken er at man har et bitset for dem man liker og et bitset for dem man misliker. Man begynner å prosessere i synkende rekkefølge fra $N-1$ og fortsetter nedover. Når A liker B vil A sitt liker bitset bli unionen av A og B sine liker bitset, og tilsvarende for misliker. Hvis A misliker B blir A sitt liker bitset unionen med B sitt misliker bitset og motsatt.

Full løsning For full løsning må man samle ideene fra subtask 2 og 3. Dette blir litt mer styrete av at innad i en strongly connected komponent vil ikke nødvendigvis alle like hverandre, så man må først finne komponentens forhold til alle andre komponenter, og så sjekke A og B sine forhold til egne komponenter.

Listing 5: C++-løsning av Mine Fienders Fiender

```cpp
#include<bits/stdc++.h>
using namespace std;
using arr = array<int, 2>;

constexpr int maxn = 5e4 + 5;

bitset<maxn> liker[maxn], misliker[maxn];

vector<arr> adj[maxn], rev_adj[maxn], scc_adj[maxn];
bool visited[maxn] = { 0 };
int scc[maxn] = { 0 };  // Gir hvilken komponent en node er i
int likt_av_komponent[maxn] = { 0 }; // Gir forholdet innad i en
    komponent


void dfs(int p, vector<int>&output)
{
    visited[p] = true;
    for (auto [i,j] : adj[p])
    {
        if (!visited[i]) dfs(i, output);
```

```
    }
    output.emplace_back(p);
}

void samle_komponent(int p, int num, int er_likt)
{
    scc[p] = num;
    likt_av_komponent[p] = er_likt;
    for (auto [i,j] : rev_adj[p])
    {
        if (scc[i]==0) samle_komponent(i, num, j * er_likt);
    }
}

int main()
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);

    int N, M, Q;
    cin >> N >> M >> Q;

    int c, a, b;
    for (int i = 0; i < M; i++)
    {
        cin >> c >> a >> b;
        adj[a].emplace_back(arr{b, c});
        rev_adj[b].emplace_back(arr{a, c});
    }

    // Bruker Kosaraju-Sharirs algoritme
    vector<int>order;
    for (int i = 0; i < N; i++)
    {
        if (!visited[i]) dfs(i, order);
    }
    reverse(order.begin(), order.end());

    int next_scc = 1;
    for (int i : order)
    {
        if (scc[i]!=0) continue;
        samle_komponent(i, next_scc++, 1);
    }

    // Lager den kondenserte grafen
    for (int i = 0; i < N; i++)
    {
        for (auto [y, j] : adj[i])
```

```
        {
            if (scc[i]==scc[y]) continue;
            scc_adj[scc[i]].emplace_back(arr{scc[y], j *
                likt_av_komponent[i] * likt_av_komponent[y]});
        }

        if (likt_av_komponent[i] == 1) liker[scc[i]][i]=true;
        else misliker[scc[i]][i] = true;
    }
    // Regner ut venneforhold
    for (int i = next_scc - 1; i >= 0; i--)
    {
        for (auto [y, j] : scc_adj[i])
        {
            if (j == 1)
            {
                liker[i] |= liker[y];
                misliker[i] |= misliker[y];
            } else
            {
                liker[i] |= misliker[y];
                misliker[i] |= liker[y];
            }
        }
    }

    while(Q--)
    {
        cin >> a >> b;
        cout << likt_av_komponent[a] * (liker[scc[a]][b] -
            misliker[scc[a]][b]) << "\n";
    }
}
```