Papirfly

Papirfly handler om å avgjøre hvor mange ganger Pia må kaste papirflyet for å komme en lengde N. Papirflyet kan fly opp til 10 meter, men ettersom man vil minimere antall kast vil vi alltid fly lengst mulig, altså 10 meter. Svaret blir da ca. $N/10$, og så vil man runde opp ettersom hver påbegynte 10 meter telles som et nytt kast, slik at vi får $\lceil N/10 \rceil$.

Listing 1: C++-løsning av Papirfly

```cpp
#include<iostream>
using namespace std;

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

    // Når man bruker heltall i C++ vil svaret bli et heltall
    // Man kan derfor lage en ceil-funksjon ved å plusse på 9 og
       runde ned
    cout << (n + 9) / 10 << "\n";
}
```

.

Runddans

I runddans er det N par i en ring. Du får vite for hver mann hvem de danser med originalt og etter at alle damene har rotert et hakk med klokken. Målet er så å si rekkefølgen til mennene med klokken.

På **Subtask 1** har vi at det er to par. Da trenger man bare å avgjøre hvem man skal skrive ut først, ettersom den andre mannen uansett blir skrevet ut som neste. Vi skal begynne med det navnet som er tidligst i alfabetet, slik at vi kan sortere mennenes navn og skrive dem ut i den rekkefølgen.

På **Subtask 2** har vi at det er tre par. Vi vet at vi skal begynne med det første navnet, så vi trenger bare å avgjøre rekkefølgen mellom de to gjennværende. Hvis vi tegner opp dette tilfellet som i **??** ser vi at neste person i rekkefølgen er han som etter rotasjonen danser med hun som før rotasjonen danset med første person. Vi kan da løse subtasken ved å først finne første man i alfabetet. Så sjekker vi hvem som han danset med først, og neste mann i rekkefølgen blir han som danset med henne etterpå. Kan til slutt skrive ut siste man som ikke danset med hun som første man danset med.

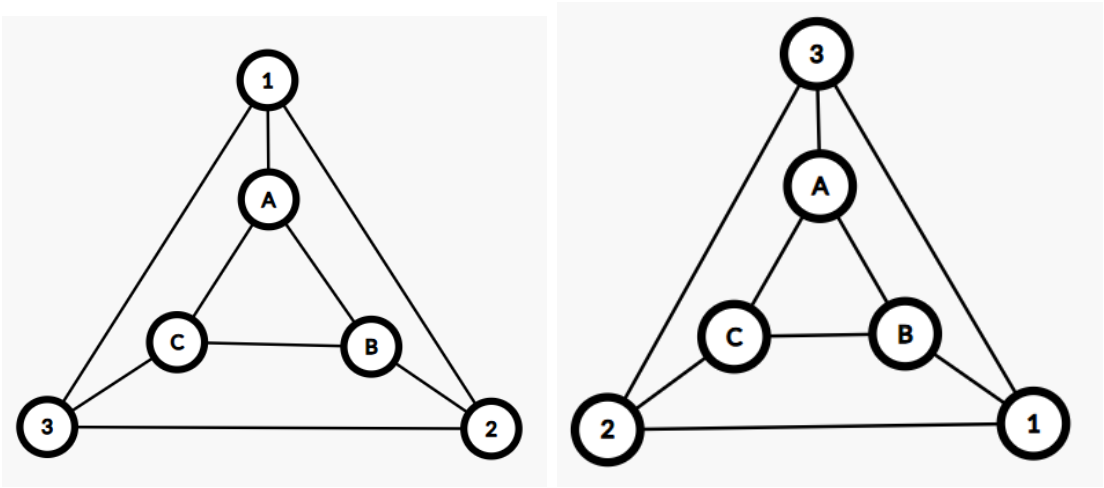

Figure 1: Figur før (venstre) og etter (høyre) rotasjonen når $N = 3$. A, B og C er nå mennene, mens 1, 2 og 3 er kvinnene. Parene er når en bokstav og et tall er koblet sammen.

For **full løsning** ser vi at mønsteret fra Subtask 2 forplanter seg videre. Når vi skal finne neste mann kan vi ta å se hvem forrige startet å danse med, og så vil det da han som danset med henne etterpå være nestemann. Slik kan vi fortsette til vi har kommet rundt sirkelen. Spørsmålet blir da hvordan man skal holde styr på

hvem som er neste. Vi trenger for hver mann å finne ut hvem de danset med først og så finne ut hvem kvinnen danset med til slutt. Dette kan løses med et map for menn og et for kvinner.

Listing 2: C++-løsning av Runddans

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

int main()
{
    int N;
    cin >> N;

    map<string, string> kvinner, menn;

    string m, k1, k2;
    for (int i = 0; i < N; i++)
    {
        cin >> m >> k1 >> k2;
        menn.insert({m, k1});
        kvinner.insert({k2, m});
    }
    // Finner mannen tidligst i alfabetet
    string pos = min_element(menn.begin(), menn.end())->first;

    for (int i = 0; i < N; i++)
    {
        cout << pos << "\n";
        pos = kvinner.at(menn.at(pos));
    }
}
```

.

Jobbjakt

I Jobbjakt får man tilbud om lønn fra en uniform fordeling fra l til r. I hver serie kan man bare godta et tilbud, og målet er å få høyest mulig lønn totalt. Her er noen forskjellige strategier:

10 poeng Godta alltid første tilbud. Dette gir i gjennomsnitt $(r-l)/2$.

59 poeng Gå gjennom den første tredjedelen og så velge neste som er bedre enn den beste man har sett til nå eller ta den siste. Dette er en strategi som funker selv om man ikke kjenner l og r, men ettersom vi kjenner dem kan man gjøre bedre.

84 poeng Det kan vises at hvis man har n trekkninger fra en uniform fordeling fra l til r vil forventningsverdien til max av trekningene være $l+(r-l)N/(N+1)$. Man kan da lage en strategi ved å stoppe hvis man finner noe bedre enn det. Her burde N være antall tilbud igjen.

Listing 3: C++-løsning som gir 84 poeng på Jobbjakt

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

void solve()
{
    int n, l, h, a;
    cin >> n >> l >> h;

    while (n--)
    {
        cin >> a;
        if (n * a >= (n-1) * h + l || n == 0)
        {
            cout << "ja" << endl;
            break;
        }
        else {
            cout << "nei" << endl;
        }
    }
}

int main()
{
    int r;
    cin >> r;
    while (r--) solve();
}
```

.

91 poeng [full løsning] For å få maks uttelling må man regne litt på forventningsverdier. Første oppdagelse er at hvis $N = 1$ vil man gjennomsnitlig få $(l + h)/2$. For å slippe ekstra notasjon kan vi tenke på det som den beste halvdel eller 0.5 av intervallet. Vi kan da skrive at $E[1] = 0.5$, altså at hvis vi får et tilbud vil forventningsverdien være i midten av intervallet. Men hva så når $N > 1$? For $N = 2$, ser vi at hvis første tilbud er lavere enn $E[1]$ vil vi takke nei, mens hvis tilbudet er bedre enn $E[1]$ vil vi takke ja. Tilfellet vi takker nei vil skje med sannsynlighet $E[1]$ og ha forventningsverdi $E[1]$, slik at det leddet bidrar $E[1]^2$. Vi vil takke ja med sannsynlighet $(1 - E[1])$ og da vil forventningsverdien være gjennomsnittet av $E[1]$ og h. Det gir oss $E[2] = E[1]^2 + (1 - E[1])(E[1] + h)/2$. Dette gjelder generelt og kan omskrives til:

$$E[N] = \frac{1 + E[N-1]^2}{2}$$

hvor $E[N]$ er forventet plassering i intervallet med N tilbud igjen. Ettersom man hare basecasen $E[1] = 0.5$, kan man da regne dette ut for alle $N \in \mathbb{N}$. For å få en verdi å sammenligne med kan man teste om $tilbud \geq l + (h - l)E[N]$ når man har N forslag igjen etter dette.

Listing 4: Optimal C++-løsning for Jobbjakt

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

constexpr int NMAX = 20;
vector<double>E(NMAX + 1, -1);

void solve()
{
    int n;
    double l, h, a;
    cin >> n >> l >> h;

    while (n--)
    {
        cin >> a;

        if (n == 0 || a >= E[n] * (h - l) + l)
        {
            cout << "ja" << endl;
            break;
        } else {
            cout << "nei" << endl;
```

```
        }
    }
}

int main()
{
    E[0] = 0, E[1] = 0.5;
    for (int i = 2; i <= NMAX; i++) E[i] = (1 + E[i-1]*E[i-1]) /
       2;

    int r;
    cin >> r;
    while (r--) solve();
}
```

Uforutsigbare ormhull

I Uforutsigbare ormhull blir vi spurt om å beregne optimal forventet reisetid fra A til B i en graf. Grunnen til at det er forventet reisetid er at det noen steder er ormhull man kan bruke som gjør at du blir teleportert til et annet ormhull. Når du kommer ut av et ormhull må du gå inn i et annet ormhull, før du kan gå inn i det samme igjen.

Subtask 1 Her er det ingen ormhull, så det er bare å finne korteste vei. Dette kan gjøres med Dijkstras algoritme, som kommer til å være essensiell i hele oppgaven.

Observasjon 1 For hvert ormhull har man et valg. Hvis man kommer ut derfra, vil man gå direkte til nærmeste ormhull eller direkte til mål.

Subtask 2 Nå er det to ormhull. Da har man to muligheter: enten dra direkte til mål (hvis mulig) eller dra via ormhullene (hvis mulig). Man velger den som er optimal. Man kan starte med å beregne avstand fra start, slutt og ormhullene til alle andre noder, uten å gå gjennom et ormhull. Dette gjør det enklere å sette opp forventningsverdiene etterpå. Forventningsverdiene for å gå utenom (uten) ormhull og via (med) ormhull er:

$$E[uten] = [A, B]$$
$$E[med] = \min([A, P], [A, Q]) + [Q, B]/2 + \min([P, B], 2[P, Q] + [Q, B]/2)$$

og forklaring av formelen kommer videre. [A, B] betyr her avstanden fra A til B, uten å gå gjennom et ormhull, P og Q er de to ormhullene hvor Q er det ormhullet som har minst eller lik avstand til B. $E[uten]$ avstanden fra A til B, mens det er litt mer sammensatt å regne ut $E[med]$. Første leddet er minste avstand til et ormhull. Andre leddet er tilfellet at du kommer ut av ormhull Q og siden det er nærmest B vil du gå til B direkte. Siste leddet er at du kommer ut av P, og da er det enten optimalt å gå direkte til mål eller til Q. Hvis det er optimalt å gå til Q får vi at $E[P] = [P, Q] + \frac{1}{2}[P, Q] + \frac{1}{4}[P, Q] + \cdots + [Q, B]$. Dette blir en geometrisk rekke, siden du potensielt kan bli teleportert tilbake til P uendelig med ganger, men hver gang med mindre sannsynlighet. Rekken blir til $E[P] = 2[P, Q] + [Q, B]$, hvor $E[P]$ er forventet avstand fra P til B. Forslag til kode for denne subtaksen er under.

Observasjon 2 Forventet avstand når du går inn i et ormhull er lik uavhengig av hvilket ormhull det er. Forventet avstand for en node er da minimum av (avstand til mål, avstand til nærmeste lovlige ormhull + avstand fra et generelt ormhull til mål).

Hvordan regne forventningsverdi i det generelle tilfellet Vi kommer til å bruke observasjon 1 og 2 til å løse dette. Vi vet at for hvert ormhull har vi et valg om man skal gå videre til mål eller til et nytt ormhull. For å holde styr på disse to valgene kan vi si at de ormhullene j som går direkte er i X mens de som går til et nytt ormhull er i Y. Det gir følgende formel for forventningsverdi for å gå inn i et generelt ormhull $E[P]$: (Merk at $[j, W]$ vil være avstand fra j til nærmeste ormhull som vi kaller W, $|P|$ er antall ormhull og $|Y|$ er antall ormhull som ikke går direkte til mål)

$$E[P] = \sum_{j \in X} \frac{1}{|P|}[j, B] + \sum_{j \in Y} \frac{1}{|P|}\left([j, W] + E[P]\right)$$

Dette kan forenkles til:

$$|P| \cdot E[P] = |Y| \cdot E[P] + \sum_{j \in X}[j, B] + \sum_{j \in Y}[j, W]$$

og hvis man løser for $E[P]$:

$$E[P] = \frac{\sum_{j \in X}[j, B] + \sum_{j \in Y}[j, W]}{|P| - |Y|}$$

Altså hvis vi har valgt hvilke ormhull som går direkte og de som går videre kan vi regne ut forventet avstand for å gå inn i et ormhull. For å velge dette kan vi starte med å anta at alle ormhullene er i X og så sortere dem basert på hvor mye summen vil synke ved å flytte dem over og så flytte dem over en etter en til Y. Dette kan gjøres grådig hvis man vet avstanden fra alle ormhull til nærmeste andre ormhull og til mål.

Gruppe 5 Her kan man kjøre dijkstra fra hvert ormhull for å få all den informasjonen om avstander man trenger. Etter det beregner man forventningsverdi på måten over.

Full løsning Her har man ikke lenger muligheten til å kjøre dijkstra fra hvert ormhull. Vi må derfor finne en annen måte å få korteste avstand mellom et ormhull og et annet ormhull. Det er flere måter å gjøre det på og her er en av dem. Vi legger alle ormhullene til som startnoder. Hvis vi nå kjører dijkstra får vi korteste avstand til et ormhull for alle andre noder enn ormhullene selv. For å få korteste avstand for ormhullene lar vi hver node bli utforsket av de to nærmeste ormhullene. Hvis vi da ser på et ormhull vil den ha avstanden til nærmeste ormhull (seg selv) og nærmeste nabo. For å gjøre dette kan vi legge på et ekstra lag i grafen vi kjører dijkstra i og passe på logikken for å huske på hvilket ormhull vi startet i. En full løsning som gjør dette er under koden for subtask 2.

Listing 5: C++-løsning av subtask 2 på ormhull

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

constexpr ll inf = (1ll << 60);
constexpr int maxn = 2e5;

vector<arr> adj[maxn];
vector<int>ormhull(2);

// Indeks 0 er start, 1 er slutt, 2 og 3 er ormhull
vector<ll> dist[4];

void dijkstra(int p, const int ind)
{
    ll cost = 0;

    priority_queue<arr, vector<arr>, greater<arr>> pq;
    pq.push({cost, p});
    dist[ind][p] = 0;

    while (!pq.empty())
    {
        auto [cost, p] = pq.top();
        pq.pop();

        if (dist[ind][p] != cost) continue;

        for (auto [t, i] : adj[p])
        {
            if (dist[ind][i] > cost + t)
            {
                dist[ind][i] = cost + t;

                // Sikrer at man ikke går videre fra en node som
                    er et ormhull
                if (count(ormhull.begin(), ormhull.end(), i) == 0)
                    pq.push({cost + t, i});
            }
        }
    }
}

int main()
```

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

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

    if (k != 2) return 0;

    int u, v, t;
    for (int i = 0; i < m; i++)
    {
        cin >> u >> v >> t;
        adj[u].push_back({t, v});
        adj[v].push_back({t, u});
    }

    for (int &i : ormhull) cin >> i;

    // Forbereder dijkstra
    dist[0].assign(n + 1, inf);
    dist[1].assign(n, inf);
    dist[2].assign(n, inf);
    dist[3].assign(n, inf);

    dijkstra(a, 0);
    dijkstra(b, 1);

    if (dist[1][ormhull[0]] > dist[1][ormhull[1]]) swap(ormhull
        [1], ormhull[0]);

    dijkstra(ormhull[0], 2);
    dijkstra(ormhull[1], 3);

    ld svar = min(
        (ld)dist[0][b],
        (ld)min(dist[0][ormhull[0]], dist[0][ormhull[1]])+      //
            avstand til nærmeste ormhull
        0.5*dist[2][b] +                                        //
            ormhullet nærmest mål
        0.5*min(dist[3][b], dist[2][b] + 2*dist[3][ormhull[0]]) //
            dra til ormhullet, eller til mål
    );

    if (svar * 2 < inf) cout << fixed << setprecision(15) << svar
        << endl;
    else cout << -1 << endl;
}
```

.

Listing 6: Full C++-løsning av ormhull

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;
using ld = long double;
using arr = array<ll, 2>;
using arrr = array<ll, 3>;

constexpr int maxn = 2e5 + 5;
constexpr ll inf = 1ll << 58;
vector<arr> adj[maxn];

ll dist[2][maxn];
arr ormhull_dist[2][maxn];

bool is_portal[maxn] = { 0 };

// lst er bare hvilken liste man legger det i, for kaller denne
   funksjonen flere ganger med ulikt startpunkt
void dijkstra(int lst, int start)
{
    priority_queue<arr, vector<arr>, greater<arr>> pq;
    pq.push({0, start});
    dist[lst][start] = 0;

    while (!pq.empty())
    {
        auto [c, p] = pq.top();
        pq.pop();

        if (is_portal[p]) continue;

        for (auto &i : adj[p])
        {
            if (c + i[1] < dist[lst][i[0]])
            {
                dist[lst][i[0]] = c + i[1];
                pq.push({c + i[1], i[0]});
            }
        }
    }
}

void modifisert_dijkstra(vector<int> start)
{
    priority_queue <arrr, vector<arrr>, greater<arrr>> pq;
```

```
// Legger til alle ormhull
for (int i : start)
{
    pq.push({0, i, i});
    ormhull_dist[0][i] = {0, i};
}

while (!pq.empty())
{
    auto [c, p, par] = pq.top(); // avstand, posisjon,
       forelder
    pq.pop();

    for (const arr &i : adj[p])
    {
        if (ormhull_dist[0][i[0]][1] == 0)
        {
            // Hvis noden ikke har blitt utforsket
            ormhull_dist[0][i[0]] = {c + i[1], par};
            pq.push({c + i[1], i[0], par});
        }
        else if (c + i[1] < ormhull_dist[0][i[0]][0] &&
           ormhull_dist[0][i[0]][1] == par)
        {
            // Hvis du finner en billigere vei med start i
               samme node
            ormhull_dist[0][i[0]] = {c + i[1], par};
            pq.push({c + i[1], i[0], par});
        }
        else if (c + i[1] < ormhull_dist[0][i[0]][0])
        {
            // Hvis du finner en billigere vei enn det
               billigste alternativet
            swap(ormhull_dist[0][i[0]], ormhull_dist[1][i[0]])
               ;
            ormhull_dist[0][i[0]] = {c + i[1], par};
            pq.push({c + i[1], i[0], par});
        }
        else if (ormhull_dist[0][i[0]][1] != par &&
            (ormhull_dist[1][i[0]][1] == 0 || c + i[1] <
               ormhull_dist[1][i[0]][0]))
        {
            // Hvis du finner en billigere vei enn det nest
               beste alternativet
            ormhull_dist[1][i[0]] = {c + i[1], par};
            pq.push({c + i[1], i[0], par});
        }
    }
```

```
    }
}

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

    int n, m, p, a, b;
    cin >> n >> m >> p >> a >> b;
    a++,b++;

    ll u, v, c;
    for (int i = 0; i < m; i++)
    {
        cin >> u >> v >> c;
        u++,v++;

        adj[u].emplace_back(arr{v, c});
        adj[v].emplace_back(arr{u, c});
    }
    vector<int> ormhull(p);
    for (int &i : ormhull)
    {
        cin >> i;
        i++;
        is_portal[i] = true;
    }

    for (int i = 1; i <= n; i++)
    {
        dist[0][i] = dist[1][i] = inf;
        ormhull_dist[1][i][0] = ormhull_dist[0][i][0] = inf;
    }

    dijkstra(0, a);
    dijkstra(1, b);
    modifisert_dijkstra(ormhull);

    // Her begynner utregning av forventningsverdi
    // Det er litt knotete fordi man har med uendelig å gjøre, med
        det er i hvert fall en måte å gjøre det på

    sort(ormhull.begin(), ormhull.end(), [] (const int &lhs, const
        int &rhs) {
        return (ormhull_dist[1][lhs][0] - dist[1][lhs]) < (
            ormhull_dist[1][rhs][0] - dist[1][rhs]);
    });
```

```
    ld best = inf;
    ld sum_x = 0, sum_y = 0;
    int infinities = 0;

    if (p > 1)
    {
        for (int i : ormhull)
        {
            if (dist[1][i]== inf) infinities++;
            else sum_x += dist[1][i];
        }
        ld dp = p;
        if (infinities == 0) best = sum_x / dp;

        for (int i = 0; i < p - 1; i++)
        {
            if (dist[1][ormhull[i]] == inf) infinities--;
            else sum_x -= dist[1][ormhull[i]];

            if (ormhull_dist[1][ormhull[i]][0] == inf) infinities
               ++;
            else sum_y += ormhull_dist[1][ormhull[i]][0];

            if (infinities == 0)
            {
                best = min(best, (sum_x + sum_y) / (dp - (ld)(i +
                   1)));
            }
        }
    }

    best = min(best + ormhull_dist[0][a][0], (ld)dist[1][a]);

    if (best >= inf) cout << "-1\n";
    else cout << fixed << setprecision(15) << best << "\n";
}
```

.

Belysning

Subtask 1 Her har vi at $b_j \geq 2a_j$, noe som betyr at det vil aldri lønne seg å bygge oppover framfor å lage nye gatelykter. Hvis du hever en gatelykten et steg kan du potensielt dekke to nye ruter, men de kunne du også ha dekket ved å plassere nye gatelykter på de to rutene. Man trenger derfor å finne minste antall gatelykter med høyde 0 som trengs for å lyse opp hele gaten. En måte dette kan gjøres er at man tar høyeste ikke enda belyste punkt og plasserer en gatelykt. Etterpå sjekker du om du belyste andre punkter. Dette kan gjøres i $O(N^2)$ som er godt innenfor når $N = 100$. Alle spørsmål vil ha samme antall lykter, så da ganger man bare med a_j for hvert spørsmål.

Subtask 2 Her kan man prøve litt forskjellige brute force fremgangsmåter. Siden N er liten er det et ganske begrenset antall høyder som vil gi mening for et punkt, så man kan prøve ut alle sammen for alle punkter og gjøre det også for alle spørsmål.

Subtask 3 Denne subtasken kommer til å hinte litt om hvordan den fulle løsningen vil være. Her er det mye dyrere å lage nye lyktestolper enn å gjøre dem høyere. Siden $N \leq 100 < 1000 < a_j$ vil det være optimalt å bare bygge en lyktestolpe totalt. Spørsmålet er så hvor det er best å lage den. Her går det an å bare prøve å se i alle punktene og finne den som krever lavest høyde for å dekke alle andre.

Hint for full løsning I forrige subtask visste vi at det var optimalt med kun en lyktestolpe, og det forenklet problemet en god del. Kan vi prøve å generalisere dette. Merk også at ettersom $N = 100$ og $Q = 10000$ virker $O(N^2Q)$ til å være litt tregt. Går det ann å dele opp slik at vi gjør litt preprosessering først og så svarer på spørsmål på under $O(N^2)$ tid per spørsmål?

Full løsning Ja, det gjør det. Tanken er at vi klarte å finne en optimal løsning når det bare var en lyktestolpe, og det kan så hjelpe oss med å regne ut hva som er optimalt for to lykte stolper. Når vi skal finne laveste høyde vi trenger for å dekke hele gaten med to lyktestolper kan vi tenke oss at vi deler gaten opp i to deler, feks før og etter index i. Så må vi finne laveste høyde for å dekke første interval og andre interval, og hvis vi da prøver alle i-er fra start til slutt vil vi ha funnet billigste måte dekke gaten med to lyktestolper. Dette kan så generaliseres videre. Hvis vi regner ut billigste måte å dekke enhver prefix av gaten med N lyktestolper, kan vi ganske enkelt regne ut det samme for $N+1$ lyktestolper. Vi sjekker bare for alle prefixer med N lyktestolper hvor mye det koster å dekke resten av gaten med 1 lyktestolpe, og velger den som gir minst sum. Ved å være litt strategisk med hvordan man gjør dette kan dette gjøres i $O(N^3)$ tidskompleksitet som vist under. Når man så har regnet ut hvor mye høyde man må legge til hvis man har et gitt antall lykte stolper, kan man for hvert spørsmål sjekke hvilket antall lyktestolper som er optimalt i $O(NQ)$ tid, slik at vi totalt får $O(N^3 + NQ)$.

Listing 7: Full C++-løsning for belysning

```cpp
#include<bits/stdc++.h>
using namespace std;
constexpr int inf = 1e9;

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

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

    vector<int> y(N), left_skew(N), right_skew(N);
    for (int& i : y) cin >> i;
    for (int i = 0; i < N; i++) right_skew[i] = y [i] + i,
       left_skew[i] = y[i] + (N - 1 - i);

    vector<vector<int>> max_right(N, vector<int>(N, -1)), max_left
       (N, vector<int>(N, -1));
    for (int i = 0; i < N; i++)
    {
        max_right[i][i] = right_skew[i];
        max_left[i][i] = left_skew[i];
        for (int y = i + 1; y < N; y++)
        {
            max_right[i][y] = max(max_right[i][y-1], right_skew[y
               ]);
            max_left[i][y] = max(max_left[i][y-1], left_skew[y]);
        }
    }

    vector<vector<int>> intervals(N, vector<int>(N, inf));
    vector<vector<int>> min_cost(N + 1, vector<int>(N, inf)); // [
       antall lyktestolper, posisjon]

    // Beregner minste høyde man trenger for å dekket et intervall
        med en lyktestolpe
    for (int j = 0; j < N; j++)      // Start av intervallet
    {
        for (int i = j; i < N; i++) // Slutt av intervallet
        {
            for (int y = j; y <= i; y++)
            {
                intervals[j][i] = min(
                    intervals[j][i],
                    max(max_left[j][y] - left_skew[y], max_right[y
```

```
                        ][i] - right_skew[y])
                );
            }
        }
    }

    // Første rad
    for (int i = 0; i < N; i++) min_cost[1][i] = intervals[0][i];

    // Resten av radene
    for (int p = 2; p <= N; p++) // Antall lyktestolper
    {
        for (int i = 0; i < N; i++) // Hvor lang prefix
        {
            for (int y = 0; y < i; y++)
            {
                min_cost[p][i] = min(
                    min_cost[p][i],
                    min_cost[p - 1][y] + intervals[y + 1][i]
                );
            }
        }
    }

    int a, b, output;
    while (Q--)
    {
        cin >> a >> b;

        output = inf;
        for (int i = 1; i <= N; i++) output = min(output, a * i +
           b * min_cost[i][N - 1]);

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