Lynnedslag

På **første subtask** er det bare ett lynnedslag, som betyr at alle husene som rammes ikke vil ha internett, mens alle de som ikke ble rammet vil ha internett. Antall hus som rammes er $b - a + 1$, som betyr at $N - (b - a + 1)$ hus har internett den 2. oktober.

Subtask 2 har kun lynnedslag som treffer ett hus. Her kan man holde oversikt over tilstanden til alle husene som en liste med N `True`/`False`-verdier, som til å begynne med alle er satt til `True`. For hvert lynnedslag flipper man verdien i listen på plass a_i. Til slutt går man gjennom listen og teller antall `True`-verdier.

Subtask 3 har lynnedslag som treffer intervaller. Her vil det være mulig å lage en liste slik som i forrige testgruppe, og oppdatere alle husene i hvert intervall for hvert lynnedslag. Kjøretiden blir i verste tilfelle $O(N \cdot K)$, siden hvert lynnedslag kan treffe opptil N hus. I verste tilfelle blir dette $10000 \cdot 1000$ operasjoner, som går fint, siden hver operasjon er svært liten (bytte True med False).

Subtask 4 har ingen videre begrensninger. Her kan man gjøre en observasjon om intervaller. Hvis man lagrer tallet +1 ved starten av hvert intervall, og tallet -1 ved slutten av hvert intervall, vil man på slutten av natten kunne iterere fra venstre mot høyre, og plusse sammen intervall-grensene underveis. Summen man har når man befinner seg ved et hus i, vil da være antall intervaller som har begynt til venstre for hus i, men som slutter til høyre for hus i. Dette er med andre ord antall intervaller hus i befinner seg i.

Vi vet at hus som befinner seg i et oddetall intervaller ikke har internett, mens hus som befinner seg i et partall antall intervaller har internett. Siden vi nå klarer å vite nøyaktig hvor mange intervaller ethvert hus befinner seg i, kan vi bruke pariteten til dette tallet for å telle hus som har internett, når vi gjør gjennomgangen fra venstre mot høyre.

Da ser koden omtrent sånn her ut:

```
count_change = [0]*(N+1)

for _ in range(K):
    a, b = map(int, input().split())
    count_change[a] += 1
    count_change[b+1] -= 1

interval_count = 0 # Hvor mange intervaller huset er med i
house_count = 0 # Hvor mange hus som har internett
```

```
for i in range(N):
    interval_count += count_change[i]
    # Hvis hus nummer i er truffet av et partall antall lyn
    if interval_count % 2 == 0:
        house_count += 1
print(house_count)
```

Merk at vi ikke setter `count_change[a]` lik $+1$, men heller øker den med 1. Dette er fordi flere intervaller kan starter og slutter på akkurat samme plass. Merk også at vi setter slutten, -1, til hus $b+1$, siden vi ønsker å sette forandringen til det første huset som er utenfor intervallet. For å spare oss for bryderi med at $b+1$ er utenfor lista når b lengst mulig til høyre, gjør vi plass til et ekstra, falskt hus i `count_change`. Til slutt går vi gjennom lista og øker en teller hver gang `interval_count` er et partall. I formelle termer sier vi at `interval_count` er prefix-summen av `count_change`, siden `interval_count` til hus i er lik `sum(count_change[:i+1])`.

Det siste trikset her, som mange har benyttet seg av, er at vi ikke egentlig bryr oss om antall intervaller vi er i, bare om det er et oddetall eller partall. Derfor kan `count_change` i stedet være en liste med `True`/`False`-verdier, som kun forteller oss om pariteten på antall intervaller har endret seg fra forrige hus.

Forslag til kildekode i C++ er gitt i kodeoppføring 1.

Listing 1: C++-løsning av Lynnedslag

```cpp
#include <iostream>

using namespace std;

#define MAXN 1000000

// If intervals[i] is true, that means house i has a different
   state to house i-1
bool intervals[MAXN+1];

int main() {
    int N, K;
    cin >> N >> K;
    for(int i = 0; i < K; i++) {
        int a, b;
        cin >> a >> b;
        // Mark the first house in the interval, and the first
           house after the interval
        intervals[a] = !intervals[a];
        intervals[b+1] = !intervals[b+1];
    }
```

```
    // The house "before" house 0 has internet
    bool state = true;
    int online = 0;
    for(int i = 0; i < N; i++) {
        if(intervals[i])
            state = !state;
        if(state)
            online++;
    }
    cout << online << endl;
}
```

.

Bananbonanza

På denne oppgaven må vi vurdere alle mulige måter å kombinere turer til Oslo og Bergen, og/eller å bli værende på bananlageret noen av dagene. På de ulike subtaskene er det mulig å gjøre ulike forenklinger, men vi går rett på den fulle løsningen i dette løsningsforslaget.

Siden vi ikke har tid til å teste alle kombinasjoner, må vi bruke en metode som kalles dynamisk programmering. Den baserer seg på å unngå å gjøre de samme utregningene flere ganger, ved å lagre resultatet første gang man regner det ut. I vårt tilfelle ønsker vi å lagre den største inntekten vi kan ha tjent, **og i tillegg være i tilbake i Kristiansand** på en gitt dag d. Vi definerer derfor en liste som heter `DP`, og lagrer den optimale verdien for dag d i `DP[d]`.

Men hvordan kan vi finne den optimale inntekten for en gitt dag d? Trikset er at vi regner ut `DP`-tabellen fra dag 0 og oppover, så vi kan bruke de optimale verdiene til tidligere dager. Vi har tre måter å være i Kristiansand på dag d:

- Vi var i Kristansand dagen før, $d-1$, og har ikke tjent noe mer siden da.
- Vi har nettopp kommet fra Oslo, solgte bananer i Oslo på dag $d-2$, og dro fra Kristiansand på dag $d-4$.
- Vi har nettopp kommet fra Bergen, solgte bananer i Bergen på dag $d-3$ og dro fra Kristiansand på dag $d-6$.

Listen vår med muligheter tar ikke høyde for å tilbringe flere dager i Oslo eller Bergen i strekk, men vi kan overbevise oss selv om at det aldri er en fordel å bli sittende i byen i flere dager før man selger, siden man i stedet kunne reist fra Kristansand senere.

Blant disse tre valgene, regner vi ut inntekten til hvert av dem, og velger det høyeste. Legg merke til at optimal inntekt dersom man er i Kristiansand på en gitt dag d, er avhengig av optimal inntekt hvis man er i Kristiansand på dagene $d-1$, $d-4$ og $d-6$. Disse verdiene har vi allerede, siden vi fyller inn `DP[]`-tabellen i kronologisk rekkefølge.

En siste detalj på denne oppgaven er at bilen ikke nødvendigvis trenger å være i Kristiansand på den siste dagen (dag $N-1$). Vi valgte å definere at `DP[d]` skal være den optimale inntekten av å være i Kristiansand på dag d. Dermed kan vi ikke bruke `DP[N-1]` som svaret vårt.

For å tillate at bilen ikke er i Kristiansand på siste dag, gir vi bilen 3 dager ekstra, slik at den kan komme seg hjem fra hvor som helst. For å hindre at bilen bruker

denne tiden på å selge noe ekstra bananer, setter vi salgsprisen til 0 i denne "komme seg hjem"-perioden. Svaret vårt blir dermed liggende tre dager etter siste salgsdag, i `DP[N-1+3]`.

Forslag til kildekode i C++ er gitt i kodeoppføring 2.

Listing 2: C++-løsning av Bananbonanza

```cpp
#include <iostream>

using namespace std;
using ll = long long;

#define MAXN 100000

// We are not forced to be back in Kristiansand on day N-1,
// but our DP[i] list gives us the optimal value of being there on
    day i.
// we solve this by adding 3 extra days to the problem, giving the
    car time to drive back from anywhere.
// During this extra time, the selling price of bananas is 0
int OPRICE[MAXN+3];
int BPRICE[MAXN+3];

// DP[i] is equal to the best income we can make and still be in
   Kristiansand on day i
ll DP[MAXN+3];

int main() {
    int N;
    cin >> N;
    for (int i = 0; i < N; i++)
        cin >> BPRICE[i] >> OPRICE[i];

    for(int i = 1; i < N+3; i++) {
        // One option is to just stay at the warehouse from the
           previous day
        ll best_price = DP[i-1];
        if(i >= 4)
            best_price = max(best_price, DP[i-4]+OPRICE[i-2]);
        if(i >= 6)
            best_price = max(best_price, DP[i-6]+BPRICE[i-3]);
        DP[i] = best_price;
    }
    cout << DP[N+2] << endl;
}
```

.

Personlighetstyper

Denne oppgaven var vanskeligere enn tiltenkt, siden vi hadde gjort en logisk feil, som også gjorde at fasit ble feil. De som skrev korrekte løsninger fikk ikke alle de poengene de skulle før konkurransen var over, som vi beklager på det sterkeste.

I denne oppgaven er det veldig kjekt å gjøre noe pre-prosessering slik at man på kort tid kan finne nærmeste person av hver type, fra en hvilken som helst annen person, både til høyre og venstre. Én måte å gjøre dette på er å lage en liste for hver personlighetstype, og legge inn posisjonene som har en slik person. Lista kan binærsøkes i for å finne nærmeste. For å gjøre det lettere å forholde seg til endene på lista, kan man legge til en falsk person på posisjon -1 og N, som har alle personlighetstyper samtidig.

Vi ønsker å vurdere én og én person som leder, så vi itererer fra venstre, og kaller personen vi vurderer P. Når P vurderes er allerede alle til venstre for P vurdert som ledere. For hver posisjon x i S lagrer vi `needed[x]` — hvor mange som må være ledere (blant alle vi har vurdert som ledere så langt), for å kunne lede alle personer fra posisjon 0 til og med x. Til å begynne med settes `needed[x]` til en veldig høy verdi, siden vi enda ikke har vurdert noen ledere.

Når den nye kandidaten P skal vurderes som leder, ser vi på hvor stort intervall den kan lede, ved å gå så langt til venstre og høyre vi kan, uten å møte personlighetstyper som ikke kan ledes av P. Intervallet av personer som P kan lede kaller vi $[A, B]$. På de siste subtaskene må man binærsøke eller gjøre annen preprosessering for å raskt finne A og B for enhver P.

Alle personer x som befinner seg inni dette intervallet, skal få `needed[x]` oppdatert til `needed[A-1]+1`. Dette kommer av at vi trenger å dekke alle til venstre for $[A, B]$ med ferrest mulig tidligere ledere, mens P kan dekke $[A, B]$. Dersom `needed[x]` allerede var mindre enn `needed[A-1]+1`, oppdateres den ikke likevel, siden det finnes en mer effektiv måte å dekke S fra 0 til x, som ikke bruker P som leder.

For å raskt kunne sette en min-verdi på et intervall, bruker vi et min-segmenttre med intervall-oppdateringer. Dette gjør at oppdateringer tar $\log(N)$ tid, men å lese ut en verdi tar også $\log(N)$ tid, siden vi ikke faktisk lagrer `needed[x]` direkte, men i stedet må ta min-verdien av en sti i segmenttreet for å finne `needed[x]`.

Etter man har vurdert alle personer i S til å være ledere, fra venstre mot høyre, og passet på at hver leder underveis får dekket alt til venstre for sitt intervall, i tillegg til å forsøke å oppdatere min-verdiene i sitt eget intervall, vil svaret på oppgaven ligge i `needed[N-1]`.

Forslag til kildekode i C++ er gitt i kodeoppføring 3.

Listing 3: C++-løsning av Personlighetstyper

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

// For each type, which types can they not lead?
vector<int> cantlead[26];
// what positions does each type exist at?
vector<int> type_at[26];

#define BSTLEN (1<<18)

int BST[ BSTLEN*2 ];
const int offset = BSTLEN + 1;

int BST_read(int pos) {
    pos += offset;
    int result = BST[pos];
    while(pos > 1) {
        pos /= 2;
        result = min(result, BST[pos]);
    }
    return result;
}

// left and right inclusive
void BST_update(int left, int right, int val) {
    left = offset+left-1;
    right = offset+right+1;

    while(left / 2 != right / 2) {
        if(left % 2 == 0)
            BST[left+1] = min(BST[left+1], val);
        if(right % 2 == 1)
            BST[right-1] = min(BST[right-1], val);
        left /= 2;
        right /= 2;
    }
}

int main() {
    int N, M;
    cin >> N >> M;
```

```
    cin.ignore();
    string S;
    cin >> S;

    // For each person, add it to the separate list its person
        type, allowing binary searching
    // Add a dummy person with every type at position -1 and N, to
         remove the need for bounds checks
    for(int i = 0; i < 26; i++)
        type_at[i].push_back(-1);

    for(int i = 0; i < N; i++)
        type_at[S[i]-'A'].push_back(i);

    for(int i = 0; i < 26; i++)
        type_at[i].push_back(N);

    for (int i = 0; i < M; i++) {
        char O, U; // O can't lead U
        cin >> O >> U;
        cantlead[O-'A'].push_back(U-'A');
    }

    // Now we build a min-binary segment tree, with single lookup
        and range updates.
    // It stores how many teams are needed to have teams all the
        way from the left to any position X
    // We build it as we go, so it has only considered people to
        the left of person P as leaders.
    //
    // This allows us to know exactly how many intervals are
        needed to the left of
    // an interval lead by person P, to fully cover everyone up to
         a potential team lead by P.
    // The number of intervals needed to make this team a reality,
         is applied to the team interval in the min-BST.

    fill(BST, BST+BSTLEN*2, 2000000);

    for(int P = 0; P < N; P++) {
        // find out how big an interval P can lead
        int leftstop = -1;
        int rightstop = N;

        // Now use each type we can't lead to limit how big our
            interval is
        for (int type : cantlead[S[P]-'A']) {
            // find the first person of the forbidden type to the
                right of us
```

```
            auto found = upper_bound(type_at[type].begin(),
                type_at[type].end(), P);
            int nextright = *found;
            // The previous person of that type has to be the
                closest to the left of us
            int nextleft = *(--found);
            // Use the cantlead persons to limit the interval lead
                 by person P
            leftstop = max(leftstop, nextleft);
            rightstop = min(rightstop, nextright);
        }

        // Make the interval edges inclusive, instead of exclusive
        leftstop++;
        rightstop--;

        int intervals_needed = 0;
        if(leftstop != 0) // If we don't strech all the way to the
            left,
                          // we must rely on as few intervals
                              existing as needed
            intervals_needed = BST_read(leftstop-1);

        BST_update(leftstop, rightstop, intervals_needed+1);
    }

    cout << BST_read(N-1) << endl;
}
```

.

En observasjon gjort av en deltaker er at alle intervaller slutter på personen til venstre for en personlighetstype som ikke kan ledes av lederen til forrige intervall. Dermed er det bare 26 mulige steder for forrige intervall å stoppe, og én av dem må være optimal. Vi slipper dermed å bruke et min-segmenttre. Istedet finner vi den første personen av hver type som har posisjon $\geq A$. Den mest effektive måten å få dekket alle personer i S fra 0 til $A-1$, vil være å bruke et av intervallene som stopper rett før én av disse 26 mulige personene. Dette gir betraktelig kortere kode. Godt gjort!

Nettverkssikkerhet

På denne oppgaven får vi en graf der de aller aller fleste nodene har grad 0, 1 eller 2, mens de første K kan ha mye høyere grad. Graden til en node betyr altså hvor mange andre noder den er koblet til. For å gjøre det lettere å snakke om oppgaven, kaller vi de første K nodene for "stornoder", og resten for "smånoder". Observasjonen vi må gjøre er at alle smånoder danner kjeder eller sykler, siden de maks kan ha to naboer. Langs en kjede er det ganske lett å beskytte seg mot virus, siden vi bare trenger å beskytte annenhver node. Endene på en kjede av smånoder kan enten være en smånode med grad 1, eller så kan det være en stornode. Siden det maks finnes 15 stornoder, kan vi finne alle kjeder med smånoder, og gruppere dem etter hvilke stornoder den har i hver ende. Hvor mange av smånodene i kjeden som skal ha antivirus blir da avhengig av om stornodene på endene har antivirus eller ikke.

For å løse denne oppgaven er man nesten nødt til å sette seg ned og tegne opp alle mulige varianter av kjeder av smånoder, og hvordan endene av kjeden ser ut:

- En smånode som ikke er koblet til noen andre.
- En kjede av smånoder som ikke har en stornoder i endene.
- En kjede av smånoder som danner en løkke.
- En kjede av smånoder som har en stornode i den ene enden.
- En kjede av smånoder som har stornoder i begge ender.
- En kjede av smånoder som har samme stornode i begge ender.

For hver av disse tilfellene må man vurdere hvor mange av smånodene som trenger antivirus. I en generell kjede vil det være halvparten som trenger antivirus, men kjeder av oddetall lengde må enten rundes opp eller ned. I tillegg vil antall antivirus som trengs påvirkes av om stornodene i endene har antivirus.

I løsningsforlaget går til til å begynne med ut ifra at alle stornoder har antivirus, som gir oss en nedre grense på hvor mange smånoder som har antivirus. I tillegg lagrer vi hvor mange ekstra smånoder som trenger antivirus dersom en stornode k ikke hadde antivirus likevel.

For kjeder med stornoder i begge ender, må vi kanskje ha ekstra antivirus hvis én av stornodene ikke er beskyttet, mens andre kjeder kun må ha ekstra antivirus hvis begge endene er ubeskyttede stornoder.

Ved å lagre hvor mange ekstra smånoder som trenger antivirus for alle mulige par av stornoder, i en $K \times K$-matrise, kan vi faktisk forsøke alle mulige varianter av at stornoder har og ikke har antivirus, og bruke matrisen for å raskt summere opp totale antall noder med antivirus. Det er maks 15 stornoder, så det blir 2^{15} muligheter, der mange muligheter kan forkastes fort fordi stornoder som er koblet til hverandre krever at minst én av dem har antivirus. For å raskt forsøke alle 2^K muligheter, bruker vi en bitmask der hver bit forteller om stornode i er beskyttet eller ikke. Deretter kan vi bruke noen doble for-løkker for å sjekke at

- bitmasken representerer en lovlig beskytting av stornoder (ingen naboer uten beskyttelse)
- kjeder med ubeskyttede noder i én eller begge ender som krever ekstra antivirus til smånoder blir telt med.

Forslag til kildekode i C++ er gitt i kodeoppføring 4.

Listing 4: C++-løsning av Nettverkssikkerhet

```cpp
#include <iostream>
#include <vector>
#include <cassert>

using namespace std;

#define MAXN 200000
#define MAXK 15

vector<int> wired[MAXN];
bool seen[MAXN];

// Used to track connections between the first K
bool connected[MAXK][MAXK];

// For each node with wireless, we track how many more nodes
// without wireless will need antivirus, if this wireless node is
   unprotected
int cost_of_unprotected[MAXK];
// How many extra nodes without wireless will need to be protected
   ,
// if at least one of the two wireless nodes are unprocteded
int cost_of_either_unprotected[MAXK][MAXK];
// How extra many nodes without wireless will need to be protected
   , if both are unprotected,
// compared to just one being unprotected
int cost_of_both_unprotected[MAXK][MAXK];
```

```cpp
int main() {
    int N, M, K, P;
    cin >> N >> M >> K >> P;

    for(int i = 0; i < M; i++) {
        int a, b;
        cin >> a >> b;
        wired[a].push_back(b);
        wired[b].push_back(a);
    }

    // If we give antivirus to all of the wireless nodes,
    // there will be this many non-wireless nodes that need
       antivirus.
    int guaranteed_antivirus = 0;

    // Now detect all chains connected to the wireless computers
    for(int i = 0; i < K; i++) {
        seen[i] = true;
        for(int head:wired[i]) {
            if(seen[head])
                continue;

            // If the head machine also has wireless, there is no
               chain
            if(head < K) {
                connected[min(i, head)][max(i, head)] = true;
            }

            // We are working on a chain
            int prev = i;
            int len = 0; // The number of nodes in the chain that
               don't have wireless
            while(head >= K) {
                len++;
                seen[head] = true;
                if(wired[head].size() == 1) { // A chain that just
                    ends
                    head = -1;
                    break;
                }

                // Swap head to the next in the chain
                // We don't increase len unless the next head
                   doesn't have wireless
                int next = wired[head][0] + wired[head][1] - prev;
                prev = head;
                head = next;
```

```
            }

            if(head == -1) { // One-ended chain
                guaranteed_antivirus += len/2; //odd gets floored
                if (len % 2 == 1) // One-ended chains with odd
                    length depend on
                    cost_of_unprotected[i]++;
            } else { // Two ended chain (wireless node in both
                ends)
                // It's possible for i and head to be the same
                    node, but that does not affect the logic
                guaranteed_antivirus += len/2; //odd gets floored
                if (len % 2 == 1) // For odd chains, having one or
                     both ends unprotected adds one required
                    antivirus
                    cost_of_either_unprotected[min(i, head)][max(i
                        , head)]++;
                else // for even chains, both have to be
                    unprotected for us to need an extra antivirus
                    cost_of_both_unprotected[min(i, head)][max(i,
                        head)]++;
            }
        }
    }

    // Now we must handle all chains that don't have any wireless
        machines
    for (int i = 0; i < N; i++) {
        if (seen[i])
            continue;
        if (wired[i].size() == 0) {
            seen[i] = true; // Nodes that are alone need no
                antivirus
            continue;
        }
        if (wired[i].size() == 2)
            continue; // We only handle chains at first, so find
                the ends of chains

        int len = 1;
        seen[i] = true;
        int prev = i;
        int head = wired[i][0];
        while(true) {
            len++;
            seen[head] = true;
            if(wired[head].size() == 1)
                break;
            int next = wired[head][0] + wired[head][1] - prev;
```

```
            prev = head;
            head = next;
        }

        guaranteed_antivirus += len/2; //odd gets floored
    }

    // Finally handle cycles without wireless
    for (int i = 0; i < N; i++) {
        if (seen[i]) continue;

        assert(wired[i].size() == 2);

        int len = 1;
        seen[i] = true;
        int prev = i;
        int head = wired[i][0];
        while(head != i) {
            len++;
            seen[head] = true;
            int next = wired[head][0] + wired[head][1] - prev;
            prev = head;
            head = next;
        }

        guaranteed_antivirus += (len+1)/2; //odd gets floored
            ceiled
    }

    // Add the wireless connections
    for (int i = 0; i < P; i++) {
        int a, b;
        cin >> a >> b;
        connected[a][b] = true;
        connected[b][a] = true;
    }

    // Now we have all the data needed to try all different
       protections of the K wireless nodes

    int best_seen = N;

    // A bit in the mask is 1 if it has antivirus
    for (int mask = 0; mask < (1<<K); mask++) {

        // First we check if this choice of mask is even possible,
        // By finding a pair of connected machines that neither
            have antivirus
        for(int i = 0; i < K; i++) {
```

```
            if ((mask >> i) & 0x1) // If i is protected
                continue;
            for(int j = i+1; j < K; j++) {
                if ((mask >> j) & 0x1) // If j is protected
                    continue;
                // We know that neither i or j is protected
                if (connected[i][j])
                    goto try_next;
            }
        }
        {
            // Now find the antivirus cost of this configuration
            int antiviruses = guaranteed_antivirus +
               __builtin_popcount(mask);
            for(int i = 0; i < K; i++) {
                bool i_unprotected = ((mask>>i)&0x1) == 0;;
                if (i_unprotected)
                    antiviruses += cost_of_unprotected[i];
                for(int j = i; j < K; j++) {
                    bool j_unprotected = ((mask>>j)&0x1) == 0;;
                    if (i_unprotected || j_unprotected)
                        antiviruses += cost_of_either_unprotected[
                           i][j];
                    if (i_unprotected && j_unprotected)
                        antiviruses += cost_of_both_unprotected[i
                           ][j];
                }
            }

            best_seen = min(best_seen, antiviruses);
        }
        try_next:;
    }

    cout << best_seen << endl;
}
```

.

Godkjent

På **subtask 1** er treet lite nok til at det er mulig å simulere at hvert stykke arbeid som skal valideres må sendes rundt i treet. Hver node trenger kun å vite hvem de har som direkte leder, og hvem de har som underordnede, dersom de ikke er en arbeider. Husk å sortere de underordnede før du begynner å simulere, slik at du ikke trenger å lete etter den underordnede med lavest ansattnummer.

På **subtask 2** kan du egentlig bare gjøre det samme. Begrensningen på antall personer som skal godkjenne arbeid, gir også i praksis en begrensning på hvor mye simulering det blir i sum.

På **subtask 3** forsvinner muligheten for å simulere alt. Her kan hvert stykke arbeid potensielt bli sett av 200000 stykker, og det er opp til 100000 stykker arbeid. Vi må altså kunne fortelle hele subtre av gangen at de alle har sett på et stykke arbeid. Den tiltenkte metoden er å gi hver node i treet en postorder-verdi. Den finner man ved å først gjøre en DFS. I denne DFS-en bruker man en teller, som øker hver gang man forlater en node for siste gang, på vei oppover igjen. Samtidig er det lurt å lagre i hver node, størrelsen på subtreet dens. Det vil altså si noden selv, og alle direkte og indirekte underordnede.

Observasjonen vi må gjøre er at ethvert stykke arbeid vil bli godkjent av ett eller to kontinuerlige intervaller i postorder-lista, siden et arbeid også "teller" hver gang man forlater en node for siste gang i DFS-en. Grunnen til at det kan bli to intervaller, er at arbeidet kan komme til en leder L fra en underordnet X, som er langt til høyre blant Ls underordnede. Delegering begynner alltid fra venstre, så med mindre alle Ls underordnede til venstre for X også må godkjenne, vil det bli et hull mellom intervallet som starter lengst til venstre i L, og intervallet som perfekt dekker X sitt subtre. For å raskt finne noden som er aller lengst til venstre i subtreet til L, er det fint om noder også husker hva postorderen var på vei nedover i DFS-en. (Men telleren øker bare når man forlater noder for siste gang).

Når du skal gjøre oppdateringer på et intervall trenger du forresten ikke å bruke segmenttrær her, bare bruk den samme logikken som vi brukte på oppgave 1 — Lynnedslag — og sett +1 og -1 i hver ende, slik at prefix-summen av lista blir de endelige summene til hver arbeider.

På **subtask 4** trenger man å gjøre en ekstra ting for å få programmet fort nok. Problemet med forrige løsning er at det kan ta tid å finne person L. L skal altså være så høyt opp i treet at størrelsen på L sitt subtre er større enn G_i, slik at vi vet at alle som godkjenner befinner seg nedenfor L. L skal samtidig ikke være noe høyere enn dette i treet, så L må ha et barn X, som er lederen for subtreet der alle skal godkjenne. Å hoppe fra en vilkårlig arbeider opp til en forelder som

akkurat passerer grensa for subtre-størrelse, kan gjøres med binærsøking. Da må hver node ikke bare vite forelderen sin, men også 2-forelderen sin, 4-forelderen sin, 8-forelderen sin, 16- etc. opp til N. Det blir bare 18 stykker, siden avstanden opp i treet dobles for hver forelder, og treet maks er 200 000 høyt.

Tips for å lage en slik liste: en node sin 2n-forelder vil være det samme som n-forelderens n-forelder, så den er lett å finne så lenge du bygger listene til de øverste nodene først.

Forslag til kildekode i C++ er gitt i kodeoppføring 5

Listing 5: C++-løsning av Godkjent Arbeid

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <cassert>

using namespace std;

#define MAXN 200000
#define MAXM 100000
#define MAXNLOG2 18


struct Node {
        int subtree_size; // including this node
        int parents[MAXNLOG2]; // parents[0] is direct parent
        vector<int> children;

        int postfix;
        int subtree_lowest_postfix;
};
Node nodes[MAXN];

int work_intervals[MAXN+1]; // needs to be inclusivly prefix
    summed to get values

// we only care about depth for the subtask 3 assert
int calc_subtree(int node, int depth, int *postfix) {
        // Make parent edges
        int parent = node[nodes].parents[0];
        for(int i = 1; i < MAXNLOG2; i++) {
                if(parent == -1)
                        nodes[node].parents[i] = -1;
                else {
                        parent = nodes[parent].parents[i-1];
                        nodes[node].parents[i] = parent;
```

```
                }
        }

        int subtree_size = 1;

        nodes[node].subtree_lowest_postfix = *postfix;

        sort(nodes[node].children.begin(), nodes[node].children.
            end());
        for(int child:node[nodes].children) {
                subtree_size += calc_subtree(child, depth+1,
                    postfix);
        }

        // Allocate a postfix value for ourselves
        nodes[node].postfix = *postfix;
        ++*postfix;

        nodes[node].subtree_size = subtree_size;
        return subtree_size;
}

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

        nodes[0].parents[0] = -1;
        for(int i = 1; i < N; i++) {
                int parent;
                cin >> parent;
                nodes[i].parents[0] = parent;
                nodes[parent].children.push_back(i);
        }
        // Now make the tree edges and subtree sizes
        int postfix = 0;
        calc_subtree(0, 0, &postfix);
        assert(postfix == N); // Every node is in the tree

        long long eyes_needed_sum = 0;
        for(int i = 0; i < M; i++) {
                int worker, eyes_needed;
                cin >> worker >> eyes_needed;
                assert(nodes[worker].subtree_size == 1); // Only
                    workers (leaf nodes) can do work
                eyes_needed_sum += eyes_needed;

                // First we binary search until we find the
                    highest parent with no more than eyes_needed
                    subtree size
```

```
int fully = worker; // All nodes in the fully
    subtree will accept the work
while(true) {
        int p;
        for(p = 0; p < MAXNLOG2; p++) {
                int fully_parent = nodes[fully].
                    parents[p];
                if(fully_parent == -1)
                        break;
                if(nodes[fully_parent].
                    subtree_size > eyes_needed)
                        break;
        }
        // now fully's parent 2**p is one level
            too much to go
        // if p is 0, then fully is exactly on the
             edge
        if (p == 0) {
                break;
        }
        fully = nodes[fully].parents[p-1];
}

assert(fully != -1);
assert(nodes[fully].subtree_size <= eyes_needed);
// If this subtree perfectly contains enough nodes
int fully_left = nodes[fully].
    subtree_lowest_postfix;
if (nodes[fully].subtree_size == eyes_needed) {
        work_intervals[fully_left]++;
        work_intervals[fully_left+eyes_needed]--;
        continue;
}

// Otherwise, we must also include some eyes from
    neighbouring subtrees
int additional = eyes_needed - nodes[fully].
    subtree_size;
// go to the node one above it (must exist)
int above = nodes[fully].parents[0];
assert(nodes[above].subtree_size > eyes_needed);
// Either we now get two intervals, or a long one
    that includes the fully node
int above_left = nodes[above].
    subtree_lowest_postfix;
if (above_left + additional >= fully_left) {
        // One long interval
        work_intervals[above_left]++;
        work_intervals[above_left+eyes_needed]--;
```

```
                } else {
                        // Two intervals
                        work_intervals[above_left]++;
                        work_intervals[above_left+additional]--;
                        work_intervals[fully_left]++;
                        work_intervals[fully_left+eyes_needed-
                            additional]--;
                }
        }

        // Finally go through with prefix sum to turn intervals
            into values
        int sum = 0;
        for(int i = 0; i < N; i++) {
                sum += work_intervals[i];
                work_intervals[i] = sum;
        }

        // Print out how much work each node did
        for(int i = 0; i < N; i++) {
                cout << work_intervals[nodes[i].postfix] << " ";
        }
        cout << endl;
}
```

.