Brusskap

På denne oppgaven er det to observasjoner man må gjøre:

- Når man legger inn en lapp vil alt som kommer før den lappen ikke lenger ha noe å si. Den ekstra lappen kan få skapet til å ha en hvilken som helst mengde brus fra 0 til og med K.

  Dette betyr også at det alltid vil lønne seg å legge til ekstra lapper så sent som mulig.

- Etter en lapp er lagt til kan man ikke vite nøyaktig hvor mange flasker som er i skapet, men man kan holde styr på et intervall av mulige verdier.

Løsningen blir derfor som følger: Hold styr på minium og maksimum antall flasker i skapet. Ved start vil begge disse tallene være 0, siden vi vet nøyaktig antall.

Simuler flest mulig lapper. For hver lapp oppdaterer du minimum og maksimum med tallet på lappen. I tillegg må minimum aldri bli negativ, og maksimum aldri større enn K.

Simuleringen stopper når intervallet ikke lenger er et gyldig intervall, ved at minimum $>$ maksimum. Når det har skjedd, betyr det at den siste lappen som ble simulert førte til en umulig tilstand. Derfor *må* vi legge til en ekstra lapp *før* den umulige lappen.

Vi vet ikke hva som står på lappen vi legger til, men vi holder alle muligheter åpne ved å si at skapet nå har et sted mellom 0 og K flasker. Nå kan vi fortsette simuleringen, men husk at vi må begynne med å simulere den flasken som tvang oss til å legge til en lapp, siden den ekstra lappen ble lagt til før den kritiske lappen.

Hvis lappen for eksempel sier $+6$, vil det nye intervallet bli $[6, K]$, siden maksverdien har et tak. Derimot hvis lappen sier -6, vil det nye intervallet bli $[0, K-6]$, siden globalt minimum er 0.

Fortsett slik helt til alle lappene er simulert, og husk antall ganger intervallet måtte resettes til alle muligheter. Forslag til kildekode i C++ er gitt i kodeoppføring 1.

Listing 1: C++-løsning av Brusskap

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

int main() {
    int K, N; //K er størrelse på skap, N antall lapper
    cin >> K >> N;
```

```
    int minBoks = 0, maxBoks = 0, errors = 0;

    for (int i = 0; i < N; i++) {
        int diff;
        cin >> diff; // Les inn en lapp

        // Label, for å kunne repetere en lapp uten å lese inn ny
      doLapp:
        // Oppdatere mulig intervall
        minBoks += diff;
        maxBoks += diff;

        // Passe på at intervallet kun har lovlige verdier
        if (minBoks < 0) minBoks = 0;
        if (maxBoks > K) maxBoks = K;

        if (minBoks > maxBoks) { // Ingen gyldige verdier
            errors++; // Vi legger inn en lapp /før/ lappen diff

            // Etter ekstralappen er alle muligheter åpne
            minBoks = 0;
            maxBoks = K;

            // Nå kan vi håndtere lappen diff igjen,
            // den som tvang oss til å resette intervallet
            goto doLapp;
        }
    }

    cout << errors << endl;
}
```

.

Flomvarsel

Det er flere måter å løse denne oppgaven på. Én måte å tenke på er at man gir hver rute et vannnivå som er høyt nok til å dekke alle ruter (1001). Rutene med sluk får istedet et vannivå lik høyden til den ruten.

Deretter kan man sjekke alle par med naboruter. Hvis begge to har vann, men ulik høyde, må den ruten med høyest vannstand redusere vannstanden, enten til ruten har like høyt vann som naboen, eller til det ikke lenger er vann på den ruten (vannstand = høyde).

For at dette skal gå raskt nok, ønsker man å vite nøyaktig hvilke naboruter som har ubalanse mellom vannstand, og i hvilken rekkefølge man burde rette opp i ubalansen.

For å vite hvilke naboruter som har ubalanse i utgangspunktet er det nok å merke seg hvilke ruter som fikk vannstanden sin redusert fra maksimum, på grunn av at de rutene har sluk. Dette lagres i en prioritetskø med den nye vannstanden som høyde. På den måten vil man alltid studere naboene til ruter med lavest vannstand først. Dette gjør at ruter alltid reduseres ned til sin endelige vannstand, og man får en forutsigbar kjøretid, der hver rute kan ligge i prioritetskøen maks én gang.

Etter at alle slukene er håndtert og lagt inn i køen, blir køen gjennomgått, og for hver rute med redusert vannstand blir alle naboene sjekket for om også de kan få lavere vannstand. De vil forsøke å ha like lav vannstand som naboen sin, men kan ikke ha vannstand lavere enn sin egen høyde. Dersom vannstanden blir redusert legges naboen inn i køen, med sin nye vannstand som prioritet, slik at naboens naboer også kan få vannstanden sin oppdatert, med så lav vannstand som mulig.

Når hele køen er behandlet, er det lett å sjekke hver rute om vannstanden er høyere enn høyden, isåfall er den ruten flommet over. Print antall slike ruter. Forslag til kildekode i C++ er gitt i kodeoppføring 2.

Listing 2: C++-løsning av Flomvarsel

```cpp
#include <iostream>
#include <queue>
#include <algorithm>

using namespace std;

// shorthand for pair of ints
using ii = pair<int, int>;
// Greate a generic priority queue, but with smallest value first
// The default 3rd parameter is std::lower<T>, we swap it
template<typename T>
```

```cpp
using min_heap = priority_queue<T, vector<T>, std::greater<T>>;

int heights[502][502]; // The heights of each tile
int waterlevel[502][502]; // the waterlevel at each tile
// a tile can not have a higher waterlevel than its neighbour,
// except that waterlevel is always >= height.
// sinks have their waterlevel = height to start with.
// To prevent going outside the edge, the map is surrounded
// by tiles where waterlevel = 0, which never get updated

// Whenever waterlevel is decreased, add it to the queue to
   propagate
min_heap<pair<int, ii>> waterlevel_queue;

// tries decreasing the waterlevel of the given tile
// can not go further down than heights[x][y]
// if changed, the new level is also added to the queue
void try_decrease_waterlevel(int x, int y, int downto) {
    if (downto < heights[x][y])
        downto = heights[x][y];

    if (downto < waterlevel[x][y]) {
        waterlevel[x][y] = downto;
        waterlevel_queue.push({downto, {x, y}});
    }
}

int main() {
    int L, B;
    cin >> L >> B;

    for (int y = 1; y <= L; y++)
        for(int x = 1; x <= B; x++)
            cin >> heights[x][y];
    for (int y = 1; y <= L; y++) {
        for (int x = 1; x <= B; x++) {
            waterlevel[x][y] = 1001; // Higher than all tiles
            int sink;
            cin >> sink;
            if (sink)
                try_decrease_waterlevel(x, y, 0);
        }
    }

    // Propagate lowerings of waterlevel to neighbours
    while (waterlevel_queue.size()) {
        auto top = waterlevel_queue.top();
        waterlevel_queue.pop();
        int waterlevel = top.first;
```

```
        int x = top.second.first;
        int y = top.second.second;
        try_decrease_waterlevel(x-1, y, waterlevel);
        try_decrease_waterlevel(x+1, y, waterlevel);
        try_decrease_waterlevel(x, y-1, waterlevel);
        try_decrease_waterlevel(x, y+1, waterlevel);
    }

    // Remember that the map starts at (1,1), due to the border
    int not_drained = 0;
    for (int y = 1; y <= L; y++)
        for (int x = 1; x <= B; x++)
            not_drained += heights[x][y] < waterlevel[x][y];
    cout << not_drained << endl;
}
```

.

Vannkran

Denne oppgaven handler om å simulere et hundretusentalls personer og vannkraner. Den optimale måten å fordele vannkranene på er at alle som kommer til stasjonen får hver sin kran, frem til det er flere personer på stasjonen enn det er kraner. Da må den personen med mest tomrom igjen i flasken sin vente, siden det alltid vil lønne seg å la folk med lite igjen å fylle gjøre seg ferdige. Når det er L personer ved stasjonen, K kraner, og $L > K$, er det altså de K personene med minst igjen å fylle som får fylle.

Når det kommer en ny person med flaske av størrelse c, må den størrelsen sammenlignes med alle flaskene som holder på å fylles, for å se om noen av kranene har en flaske med mer enn c volum igjen. Den letteste måten å gjøre dette på er at man lagrer når flasker som fylles kommer til å fullføre. Hvis den nye flasken fullfører tidligere enn den tregeste flasken, bør man straks begynne å fylle den nye i stedet. For å vite hvilken av kranene som har senest slutt-tid på flasken sin, brukes et set som inneholder slutt-tidspunktene til alle flasker som holder på. Grunnen til at et set brukes, er at det tillater innsetting av vilkårlig verdier på $O(\log n)$ tid, men også oppdatering av største og minste element i $O(\log n)$. For å få det minste elementet i et set brukes `set.begin()`, og det største brukes `--set.end()`. (Grunnen til at man må ta `--` er at `end()` peker til elementet *bak* siste element).

Å kunne hente ut minste element er viktig for å simulere at flasker blir ferdige, og kraner ledige. Før man forsøker å legge til en flaske må man gå gjennom settet med fyllende flasker og fjerne alle flasker som er ferdige. Dette gjøres ved å sammenligne slutt-tidspunktet til flaskene som fylles, men ankomsttidspunktet til den nye flasken. Siden flasker som fylles ligger i et set, kan man bruke `begin()` for å se når flasken som tidligst blir ferdig kommer til å fullføre. Dette gjøres helt til ingen flasker i settet er ferdige.

I det øyeblikket en flaske er ferdig, vil vi sjekke om det er noen flaske som venter på ledig kran. Når det kommer en ny flaske til en full stasjon vil den enten erstatte en flaske, eller måtte vente selv. Man må altså holde orden på disse ventende flaskene, slik at en kran som blir ledig kan begynne å fylle den med minst volum. Dette gjør man ved å opprettholde en min-prioritetskø over alle ventende flasker.

For å finne ut hvor lenge noen måtte vente, sammenlign tidpunktet flasken dro, og det tidligs mulige tidspunktet den flasken kunne ha dratt, hvis den fikk fylle hele tiden. Forslag til løsning i C++ er gitt i kodeoppføring 3.

Listing 3: C++-løsning av Vannkran

```cpp
#include <iostream>
#include <set>
#include <queue>
#include <algorithm>
#include <climits>

using namespace std;

using ll = long long; // 64 bit integer, ut to 2^63 - 1
using ii = pair<int,int>;
template<typename T>
using min_heap = priority_queue<T, vector<T>, std::greater<T>>;

// pairs. first is when the bottle arrives, second is size
ii bottles[100000];

// pair, first value is finish time, second is bottle index
// finish time assumes the bottle never gets interrupted
set<pair<ll, int>> currentlyFilling;

// pair, first is amout of room left to fill in bottle, second is
    index
min_heap<ii> waitingBottles;

// The final answer, sum of time spent waiting at the filling
    station
ll totalWaitingTime;

// No more new bottles arrive at the station until `lastTime`,
// so handle any bottles that finish before that.
// This possibly starts filling bottles from the waiting queue,
// which may also finish before the given time.
// In other words, keep finishing bottles until all filling
// bottles finish later than `lastTime`.
void endBottlesDoneByNow(ll lastTime) {
    // Keep checking, as long as there are still bottles filling
    while (currentlyFilling.size()) {
        // Earliest finisher is always the begin() of the set
        auto bottle = currentlyFilling.begin();
        ll time = bottle->first;
        int index = bottle->second;

        if (time > lastTime) // This bottle is not done yet
            break;
        // Remove the bottle from filling
        currentlyFilling.erase(bottle);
```

```
        // Bottle waited time between now and the earliest
        ll earliest = bottles[index].first + bottles[index].second
            ;
        totalWaitingTime += time - earliest;

        // Start filling a potential waiting bottle, with the
            least air
        if (waitingBottles.size()) {
            auto newBottle = waitingBottles.top();
            waitingBottles.pop();
            int newAirLeft = newBottle.first;
            int newIndex = newBottle.second;
            // The bottle will finish `newAirLeft` seconds after
                we start it
            currentlyFilling.insert({time+newAirLeft, newIndex});
        }
    }
}

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

    for (int i = 0; i < N; i++) {
        int time, size;
        cin >> time >> size;
        bottles[i] = {time, size};
    }

    // Get bottles in the order they arrive (sorted by first)
    // This gives bottles new indecies, which are used from now on
    sort(bottles, bottles+N);

    // Add bottles one by one, in the order they arrive
    for (int i = 0; i < N; i++) {
        int time = bottles[i].first;
        int airLeft = bottles[i].second;
        // The time we are done filling, if we start now
        int endTime = time + airLeft;

        // Make sure none of the bottles in currentlyFilling are
            done
        endBottlesDoneByNow(time);

        // How many taps are busy?
        if (currentlyFilling.size() < K) {
            // There is an unused tap, start filling now!
            currentlyFilling.insert({endTime, i});
```

```
        } else {
            // All taps taken, check if we should replace the
               slowest
            auto slowestBottle = --currentlyFilling.end();
            ll slowestBottleEndTime = slowestBottle->first;
            int slowestBottleIndex = slowestBottle->second;
            if (slowestBottleEndTime > endTime) {
                // We should replace the bottle!

                // How much air is left in the slowest bottle
                   right now?
                ll airInSlowest = slowestBottleEndTime - time;

                // Move slowest bottle from currentlyFilling to
                   waiting
                currentlyFilling.erase(slowestBottle);
                waitingBottles.push({airInSlowest,
                   slowestBottleIndex});

                // Insert our new bottle into currently filling
                currentlyFilling.insert({endTime, i});
            } else {
                // We should not replace even the slowest bottle
                // Place the new bottle in the waiting queue
                waitingBottles.push({airLeft, i});
            }
        }
    }

    // lastly make all bottles finish before we end
    endBottlesDoneByNow(LLONG_MAX);
    cout << totalWaitingTime << endl;
}
```

.

Maratonsaft

Denne oppgaven krever at man raskt klarer å telle antall saftstasjoner det er mulig å bruke i kombinasjon med den nye stasjonen. Ved å tegne opp og sjekke er det ikke alt for vanskelig å se hvilke stasjoner som kan, og ikke kan, brukes i kombinasjon med en gitt ny stasjon.

Det aller første du gjør er å ignorere alle stasjoner på venstresiden som kommer bak det siste brukbare fotgjengerfeltet. Ved å aldri telle disse blir resten av oppgaven enklere. Når du får en slik stasjon printer du 0, og behandler neste stasjon.

La x være posisjonen til en ny (brukbar) stasjon på venstresiden. Alle andre stasjoner på venstresiden kan kombineres med denne stasjonen, i tillegg kan alle stasjoner på høyresiden som ikke kommer før det neste fotgjengerfeltet. Altså, rund x opp til nærmeste hele K, og ignorer alle stasjoner på høyre side med posisjon $< (x + K - 1) - ((x + K - 1) \mod K)$.

Når en ny stasjon med posisjon x legges til på høyresiden, kan den kombineres med alle andre stasjoner på høyresiden, samt alle stasjoner på venstre side som ikke kommer etter det siste fotgjengerfeltet som kan brukes for å være på høyresiden ved posisjon x. Altså runder du x ned til nærmeste hele K, og ignorerer alle stasjoner på venstre side med posisjon $> x - (x \mod K)$.

I stedet for å telle alle stasjoner i områder du ikke ignorerer, kan du ta antall stasjoner totalt minus antall stasjoner i det ignorerte området. For å raskt telle antall stasjoner i et intervall, kan vi representere hvert fortau som en liste med N tall, altså opptil 1 000 000 tall. Hvert tall representerer en plass på fortauet, og er 0 hvis det ikke er noen stasjon der, og 1 hvis det er en stasjon der. Til å begynne med er alle tallene 0, siden det ikke er publisert noen stasjoner enda. Når du begynner å publisere stasjoner legger du sammen intervaller på motsatt side av veien for å finne ut hvor mange stasjoner du *ikke* kan pare opp med den nye. I tillegg oppdaterer du tallet på plassen til den nye stasjonen fra 0 til 1, slik at fremtidige stasjoner kan kombineres med denne.

Problemet med den foreslåtte løsningen er at det kan være veldig lange intervaller med tall vi må legge sammen. For å få det til å gå fort har vi to alternativer. *Fenwick-trær* og *binære segmenttrær*. Du kan lese om Fenwick-trær i Fredrik Anfinsens kompendium, side 48. Løsningsforslaget bruker binære segmenttrær.

I denne oppgaven trenger vi to segmenttrær, ett for venstre, og ett for høyre side. Begge segmenttrærne må ha løvnoder for hver mulige x-posisjon. Samtidig ønsker vi at antallet noder i segmenttreet skal være lik $2^{k+1} - 1$, for en eller annen k stor nok til at $2^k \geq N$. Siden N er maks én million, kan vi velge $k = 20$.

Den nederste etasjen i BSTet inneholder selve 0/1-tallene. Tallet på plass x ender opp på plassen $bst[\text{offset} + x]$ i bst-implementasjonen, for å gjøre plass til etasjene over. I tillegg legges det til side én plass lengst til venstre som kun prukes som padding. Derfor er offset $= 2^{20} + 1$.

Nederste rad inneholder altså 2^{20} noder, der noden lengst til venstre alltid er 0, og de påfølgende nodene representerer de N plassene vi kan ha stasjoner. Deretter følger masse noder vi aldri bruker til noe, siden vi måtte runde opp til en toerpotens. Raden over inneholder 2^{19} noder, slik at hver node på raden over er summen av to noder på raden under. Slik fortsetter det også med de 2^{18} nodene på raden over der igjen, osv. Det er veldig lett å finne barna til en vilkårlig node, ved å bruke indeksene. En node med indeks i har barnet $2i$ til venstre, og $2i + 1$ til høyre.

Sum-invariansen kan derfor skrives slik:

$$bst[i] = bst[2i] + bst[2i + 1], i < 2^{20}$$

Så lenge man alltid passer på å oppdatere alle foreldrene til løvnodene man endrer fra 0 til 1, vil man kunne bruke segmenttreet for å raskt summe sammen store intervaller, siden man allerede har mange summer ferdig utregnet. Se forslag til løsning i kodeoppføring 4.

Listing 4: C++-løsning av Maratonsaft

```cpp
#include <iostream>

using namespace std;

using ii = pair<int,int>;

#define MAXM 300000
#define MAXN 1000000
ii stations[MAXM]; // {x coord, 0 for left & 1 for right }

#define BASE (1<<20)

const int offset = BASE+1;
int leftBST[BASE*2];
int rightBST[BASE*2];

// Calculate the sum of all values between the left and right
   index
// in the given BST. Both left and right are exclusive.
// left being exclusive means you must pass left=-1 to sum
   everything.
// This is handled fine, since offset=BASE+1,
```

```
// leaving one always-0-node to the very left, at bst[BASE].
int bstsum(int* bst, int left, int right) {
        left += offset;
        right += offset;

        int sum = 0;
        while(left/2 != right/2) {
                if(left % 2 == 0)
                        sum += bst[left+1];
                if(right % 2 == 1)
                        sum += bst[right-1];
                left /= 2;
                right /= 2;
        }
        return sum;
}

// Change the value at the given index in the binary segment tree.
// Also changes the sums of all segments containing the changed
   leaf node
// upwards in the segment tree
void bstupdate(int* bst, int index, int update) {
        index += offset;
        while(index != 0) {
                bst[index]+=update;
                index /= 2;
        }
}

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

        for (int i = 0; i < M; i++) {
                int x;
                char c;
                cin >> x >> c;

                stations[i] = {x, c=='V'?0 : 1};
        }

        // Any stations beyond this on the left side can safely be
            discarded
        const int lastLeftX = N - N%K;
        // Only count stations that are not discarded
        int placedStations = 0;

        for (int i = 0; i < M; i++) {
                int x = stations[i].first;
```

```
                bool onLeftSide = !stations[i].second;

                if(onLeftSide && x > lastLeftX) {
                        cout << 0 << endl;
                        continue; // Completely ignore this booth
                }

                if(onLeftSide) {
                        // Our possibilities are every other
                           station except for those on the right
                           with x < minRightX
                        int minRightX = x+K-1;
                        minRightX = minRightX - minRightX % K;

                        int sum = bstsum(rightBST, -1, minRightX);
                             // exclusive in both ends
                        cout << placedStations - sum << endl;
                        placedStations++;

                        // Update that we have a booth here now
                        bstupdate(leftBST, x, +1);
                } else {
                        // We have to ignore all left stations
                           with x > maxLeftX
                        int maxLeftX = x - x % K;

                        int sum = bstsum(leftBST, maxLeftX, N); //
                             exclusive in both ends
                        cout << placedStations - sum << endl;
                        placedStations++;

                        // Update that we have a booth here now
                        bstupdate(rightBST, x, +1);
                }
        }
}
```

.